

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

The Practice of Cloud System Administration
Designing and Operating Large Distributed Systems, Volume 2

云系统管理

大规模分布式系统设计与运营

[美] 托马斯 A. 利蒙切利 (Thomas A. Limoncelli) 斯特拉塔 R. 查卢普 (Strata R. Chalup) 著
克里斯蒂娜 J. 霍根 (Christina J. Hogan)
姚军 等译

资深云计算专家十余年经验结晶，全方位介绍大规模分布式系统的设计和运营

理论与实践相结合，不仅介绍分布式系统架构、应用和设计原则的理论知识，而且包含Google、Facebook等公司的成功案例分析，以及DevOps等全栈融合思想

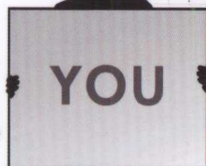
```

    .Subject = subject.String()
    change {
      notify := func(notifications map[string]error) {
        for _, n := range notifications {
          s.Notify(state, n)
        }
      }
      switch status {
        case stCritical:
          notify(a.CritNotification)
        case stWarn:
          notify(a.WarnNotification)
      }
    }
  }
  if changed {
    s.nc ← true
  }

func (s *Schedule) CheckAlert(a *conf.Alert) {
  crits := s.CheckExpr(a.Crit, stCritical, nil)
  s.CheckExpr(a.Warn, stWarning, crits)
}

func (s *Schedule) CheckExpr(a *conf.Alert, e *expr.Expr,
  if e == nil {
    return
  }
  results, _, err := e.Execute(s.cache, nil)
  if err != nil {
    log.Println(err)
    return
  }
}
Loop:
for _, r := range results {
  if a.Squelched(r.Group) {
    continue
  }
  ak := AlertKey{a.Name, r.Group.String()}
  for _, v := range ignore {
    if ak == v {
      continue Loop
    }
  }
  status[ak]

```



机械工业出版社
China Machine Press

本书是几位供职于全球最大规模技术公司的专家的力作，全方位地介绍了大规模分布式系统的设计和运营，不仅介绍有关分布式系统架构、应用和设计原则的理论知识，更有Google、Facebook等公司的成功案例分析，而且包括DevOps等全栈融合思想。细细阅读，你不仅能够了解许多分布式计算的关键概念和思想，还可以深深体会技术的发展过程，以及大规模网站运营中的经验和教训。

本书分为三部分。第一部分（第1~6章）讨论在大规模、复杂、基于云的分布式计算系统中如何进行设计的问题，从下至上逐层介绍设计的各个要素，内容涉及为实现平稳运营而应该具备的软件功能，如何选择服务平台，如何创建Web和其他应用程序的基本组件，如何通过伸缩性设计模式来扩增服务所使用的基本组件，以及如何应用弹性设计模式等。第二部分（第7~20章）描述如何运营第一部分中介绍的系统，首先介绍分布式系统运行方式以及DevOps文化、历史和实践；其次讨论如何构建服务和准备投产以及如何测试、批准和投产；然后介绍如何创建工具和自动化运营工作，以及设计文档、灾难准备；再次阐释监控的基础知识以及架构与实践；最后提出持续改善的战略。第三部分包括五个附录，涵盖运营团队的评估系统、分布式计算的历史、相关的表单模板、推荐的阅读材料以及其他参考材料。

云计算与虚拟化技术丛书

The Practice of Cloud System Administration

Designing and Operating Large Distributed Systems, Volume 2

云系统管理

大规模分布式系统设计与运营

[美] 托马斯 A. 利蒙切利 (Thomas A. Limoncelli) 斯特拉塔 R. 查卢普 (Strata R. Chalup)

克里斯蒂娜 J. 霍根 (Christina J. Hogan)

著

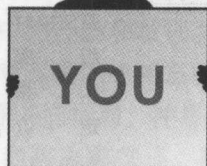
姚军 等译

```
    s.Subject = subject.String()
    s.change {
      notify := func(notifications map[string]*conf.Alert) {
        for _, n := range notifications {
          s.Notify(state, a, n)
        }
      }
      switch status {
        case stCritical:
          notify(a.CritNotification)
        case stWarn:
          notify(a.WarnNotification)
      }
    }
  }
  if changed {
    s.nc <- true
  }
}

func (s *Schedule) CheckAlert(a *conf.Alert) {
  crits := s.CheckExpr(a.Crit, stCritical, nil)
  s.CheckExpr(a.Warn, stWarning, crits)
}

func (s *Schedule) CheckExpr(a *conf.Alert, e *expr.Expr,
  if e == nil {
    return
  }
  results, _, err := e.Execute(s.cache, nil)
  if err != nil {
    log.Println(err)
    return
  }
}

Loop:
for _, r := range results {
  if s.Squelched(r.Group) {
    continue
  }
  ak := AlertKey(a.Name, r.Group.String())
  for _, v := range ignore {
    if ak == v {
      continue Loop
    }
  }
  status[ak]
```



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

云系统管理: 大规模分布式系统设计与运营 / (美) 托马斯 A. 利蒙切利 (Thomas A. Limoncelli) 等著; 姚军等译. —北京: 机械工业出版社, 2016.6

(云计算与虚拟化技术丛书)

书名原文: The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2

ISBN 978-7-111-54160-8

I. 云… II. ①托… ②姚… III. 系统管理 IV. C931.2

中国版本图书馆 CIP 数据核字 (2016) 第 154914 号

本书版权登记号: 图字: 01-2015-1918

Authorized translation from the English language edition, entitled *The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2*, 9780321943187 by Thomas A. Limoncelli, Strata R. Chalup, Christina J. Hogan, published by Pearson Education, Inc., Copyright © 2015.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2016.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括香港、澳门特别行政区和台湾地区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

云系统管理: 大规模分布式系统设计与运营

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 余洁 陈佳媛

责任校对: 董纪丽

印刷: 三河市宏图印务有限公司

版次: 2016 年 7 月第 1 版第 1 次印刷

开本: 186mm × 240mm 1/16

印张: 24.5

书号: ISBN 978-7-111-54160-8

定价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

The Translator's Words 译者序

我们这一代人是幸运的，因为我们生在变革的年代，亲眼见证了蒸汽机发明以来最重要的技术成就——计算机和互联网的兴起。

在我们儿时，连机械化都是一种梦想，仅仅在十年之前，还无法想象一部手机就可以拥有过去大型计算机的能力，过去只有在科幻小说中才能看到的情景，而今已经十分普遍。世界仿佛变得很小，一切都在我们的指尖之下，这就是科技的力量。

科技的发展帮助人类不断创造出更高质量、更低价格的产品，而产品之后的设计思想更为重要。我们今天拥有似乎无穷无尽的网络资源和计算能力，不仅是因为硬件技术的发展，更为关键的是分布式计算的出现，它将非关即开的数字化设备模型化为传统上的“模拟”系统，增进了系统的伸缩性和可靠性。借助这一关键的思路，我们才有了今天 7×24 运行的各种线上系统，以及引人遐想的“云计算”。

本书是几位供职于全球最大规模技术公司的专家的力作，全方位地介绍了大规模分布式系统的设计和运营，内容十分充实，不仅介绍了有关分布式系统架构、应用和设计原则的理论知识，更有 Google、Facebook 等公司的成功案例分析，在附录中还介绍了分布式系统的由来以及成熟度模型、设计文档等模板。尽管本书的主题是运营，但是其中系统阐述了运营与开发的关系，包括 DevOps 等全栈融合思想，对于这种环境下的开发人员也极有教益。细细阅读，不仅能够了解许多分布式计算的关键概念和思想，还可以深深体会技术的发展过程，以及大规模网站运营中的经验和教训。在艰苦的翻译期间，我们的思想也受到了很大的冲击，感受到技术发展对运营思想甚至整个企业经营思想的影响。掩卷之时也有一份担心，我们是否已经捕捉到原书的精髓，并将其准确传达给读者？

本书的翻译工作主要由姚军和陈志勇完成，徐锋、刘建林、吴兰陟、宁懿、姚红斌、白龙、陈美娜、谢志雄、方翊、陈霞、林耀成、管军凯、吴玥、余骆、耿飞等人也为翻译工作做出了贡献。

译者

2016 年 3 月

前 言 Preface

下面的表述是否真实?

- 1) 最为可靠的系统是使用廉价、不可靠的组件构建的。
- 2) Google 为数十亿用户服务所用的技术遵循的模式, 与你处理数千名用户的系统所遵循的模式相同。
- 3) 某种过程的风险越大, 你越应该尝试。
- 4) 某些最重要的软件功能是由用户从未发现的。
- 5) 你应该随意选择一些机器, 并关闭其电源。
- 6) Facebook 在接下来六个月中所要发布的代码, 其功能可能已经在你的浏览器中出现了。
- 7) 每天多次更新软件所需要的人工很少。
- 8) 随时待命并不一定是一种紧张、痛苦的体验。
- 9) 你不应该监控机器是否启动。
- 10) 运营和管理可以通过试验和证明的科学原理进行。
- 11) Google 已经对僵尸攻击时的对策进行了演练。

这些表述都是真实的, 当你读完本书, 就会知道原因。

本书讲述的是大规模云服务(数百万甚至数十亿用户使用的基于互联网的服务)的构建和运行。每天, 都有越来越多的企业采用这些技术, 因此, 本书是为所有人写的。

本书的目标读者是系统管理员和他们的经理。你无需计算机科学的背景, 但是你应该有 Unix/Linux 系统管理、联网的经验, 以及操作系统的概念。

我们重点关注构建和运营组成“云”的服务, 而不是指导云服务的使用。

云服务必须是可用的、快速的和安全的。考虑到云的规模, 这将是独特而杰出的工程成就。因此, 云规模服务设计不同于典型的企业服务。可用性很重要, 因为互联网是 7×24 开放的, 用户处于各个时区。快速的重要性在于用户会因为慢速的服务而感到沮丧, 所以缓慢的服务会被更快的竞争者所取代。安全的重要性则在于, 作为他人数据的托管者, 我们义不容辞(而且负有法律责任)地要保护人们的数据。

这些要求是互相交织的。如果网站不安全, 当然也就不可能可靠。如果网站速度不快,

可用性也就不足。如果网站下线，当然就不可能快速。

最明显的云规模服务是网站。但是，还有一个巨大的无形互联网访问服务生态系统，这些服务不是通过浏览器访问的。例如，智能手机应用通过 API 调用访问云服务。

在本书余下的部分中，我们倾向于使用“分布式计算”而非“云计算”。云计算是一个营销用语，对不同的人有不同的含义。分布式计算描述了使用许多机器（而非单一机器）提供应用程序和服务的一种架构。

本书介绍的是不受时间影响的基本原理及方法。因此，我们不推荐使用特定的产品或者技术。我们可以提供前 5 种最流行的 Web 服务器、NoSQL 数据库或者持续构建系统的对比，但是如果这么做，本书就会在出版的那一刻过时。相反，我们讨论的是选择这些系统时应该注意的特质，提供一种工作模式。这种方法的意图是帮助你在技术不断变化的漫长职业生涯中始终做好准备。当然，我们将用具体的技术和产品说明我们的观点，但是不代表我们支持这些产品和服务。

本书有时看似理想化，这是故意为之。我们想为读者提供事物将会如何发展、该坚持什么原则的愿景，目的是更上一层楼。

关于本书

本书分为两个部分——设计和运营。

第一部分捕捉我们在大规模、复杂、基于云的分布式计算系统设计上的想法。在引言之后，我们从下向上逐层介绍设计的每个要素。我们从系统管理员（而非计算机科学家）的角度介绍分布式系统，要运营一个系统就必须理解其内部原理。

第二部分描述如何运营这些系统。前面几章介绍最基本的问题。后面几章深入更为复杂的技术活动，然后是概要规划和将以上要素组合起来的战略。

最后是附加材料，包括运营团队的评估系统、被歪曲的分布式计算历史、正文中提及的表单模板、建议阅读的材料以及其他参考材料。

我们满怀兴奋之情介绍本系列书籍中的一个新特征：运营评估系统。这个系统由一系列评估活动组成，你可以用它们评估自己的运营，找出需要改进的领域。评估问题和“搜索”建议可以在附录 A 中找到。第 20 章是该系统的说明。

致谢

没有我们所在社区及来自全球的帮助及反馈，本书就不可能出版。DevOps 社区慷慨地提供了帮助。

首先，我们要感谢我们爱人和家人：Christine Polk、Mike Chalup、Eliot 和 Joanna Lear。他们的爱和耐心成就了一切。

之所以我们能够看得更远，那是因为我们站在巨人的肩上。某些章节在很大程度上得到了下列人士的支持和建议：John Looney 和 Cian Synnott（第 1 章）；Marty Abbott 和 Michael Fisher（第 5 章）；Damon Edwards、Alex Honor 和 Jez Humble（第 9 章和第 10 章）；John Allspaw（第 12 章）；Brent Chapman（第 15 章）；Caskey Dickson 和 Theo Schlossnagle（第 16 章和第 17 章）；Arun Kejariwal 和 Bruce Yan（第 18 章）；Benjamin Treynor Sloss（第 19 章）；Geoff Halprin（第 20 章和附录 A）。

感谢 Gene Kim “有策略”的启发和鼓励。有几十位人士帮助过我们——有些人提供奇闻轶事，有些人审核某些部分或者整本书。感谢他们所有人的公平方式是按照字母顺序排列，并预先向我们遗漏的所有人道歉：Thomas Baden、George Beech、Raymond Blum、Kyle Brandt、Mark Burgess、Nick Craver、Geoff Dalgas、Robert P. J. Day、Patrick Debois、Bill Duane、Paul Evans、David Fullerton、Tom Geller、Peter Grace、Elizabeth Hamon Reid、Jim Hickstein、Zachary Hueras、Matt Jones、Jennifer Joy、Jimmy Kaplowitz、Daniel V. Klein、Steven Levine、Cory Lueninghoener、Shane Madden、Jim Maurer、Stephen McHenry、Dinah McNutt、Scott Hazen Mueller、Steve Murawski、Mohit Muthanna、Lenny Rachitsky、Amy Rich、Adele Shakal、Bart Silverstrim、Josh Simon、Joel Spolsky、Desiree Sylvester、Win Treese、Todd Underwood、Nicole Forsgren Velasquez 和 Dave Zwieback。

最后（但并非不重要），感谢 Addison-Wesley 的所有人，特别要感谢 Debra Williams Cauley 将我们引见给 Addison-Wesley，并在整个过程中为我们掌舵；感谢 Michael Thurston 对本书初稿的编辑和改进，使之更加完美；感谢 Kim Boedigheimer 的协调工作以及在我们惊慌的时候帮助我们保持镇静；感谢我们的 LaTeX 奇才 Lori Hughes；感谢产品经理 Julie Nahil；感谢文字编辑 Jill Hobbs；还要感谢 John Fuller 和 Mark Taub 忍受我们的特殊要求！

第一部分 设计：构建系统

第 1 章 分布式世界中的设计

概述分布式系统的设计。

第 2 章 为运营而设计

为了实现平稳运营而应该具备的软件功能。

第 3 章 选择服务平台

物理机和虚拟机，私有云和公共云。

第 4 章 应用程序架构

创建 Web 和其他应用程序的基本组件。

第 5 章 伸缩性设计模式

扩增服务所用的基本组件。

第6章 弹性设计模式

创建可幸免于故障的系统的基本组件。

第二部分 运营：运行系统

第7章 分布式世界中的运营

分布式系统运行方式概述。

第8章 DevOps 文化

DevOps 文化、历史和实践简介。

第9章 服务交付：构建阶段

如何构建服务和准备投产。

第10章 服务交付：部署阶段

服务如何测试、批准和投产。

第11章 升级运行中的服务

如何在不停机的情况下升级服务。

第12章 自动化

创建工具和自动化运营工作。

第13章 设计文档

书面交流设计和意图。

第14章 随时待命

处理异常情况。

第15章 灾难准备

通过规划和实践强化系统。

第16章 监控基础知识

监控术语和策略。

第17章 监控架构与实践

监控组件和方法。

第18章 容量规划

在需要之前规划并提供附加资源。

第19章 建立 KPI

通过计量和反思科学地推动行为。

第20章 卓越运营

持续改善的战略。

第三部分 附录

附录 A 评估

附录 B 分布式计算和云的起源及未来

附录 C 伸缩性术语和概念

附录 D 模板和示例

附录 E 推荐读物

后记

最后的一些想法。

参考文献

Kim “有策略”的编写和出版。有几十位人士提交给 O’Reilly 一章内容，我们提供奇闻轶事，有些人审核某些部分或者整本书。余前因灾降步更，英文的 O’Reilly 按照字母顺序排列，并预先向我们道谢的所有人道歉：Thomas 编辑时曾付交我第 1 章 Eyal Blum、Kyle Brandt、Mark Burgess、Nick Craver、（为提供第 1 章和第 2 章）J. Day、Patrick Debois、Bill Duan、Paul Evans、David Finkelstein、（为提供第 3 章）Elizabeth Hamon Reid、Jim Hickstein、Zachary Huang、（为提供第 4 章）Jimmy Kaplowitz、Daniel V. Klein、Steven Levine、Cory Lee、（为提供第 5 章）Jim Maurer、Stephen McHenry、Dinah McNulty、（为提供第 6 章）Adam Muthanna、Lenny Rachitsky、Amy Rick、（为提供第 7 章）Joel Spolsky、Desires Sylvester、Win Trower、（为提供第 8 章）和 Dave Zwieback。

最后（但并非不重要），感谢 Addison-Wesley 的所有人，因编辑长时期间交面给 William Cauley 将我们引见给 Addison-Wesley，并在整个过程中为本书提供建议和反馈。感谢 Kim Bogard 对本书初稿的编辑和改进，使之更加完美；感谢 Kim Bogard 以及在我们惊慌的时候帮助我们保持镇静；感谢我们的 LaTeX 专家 Julie Nahil；感谢文字编辑 Jill Hobbs；还要感谢 John 和 Susan 为本书提供他们的特殊要求！

第一部分 设计：构建系统

第 1 章 分布式系统中的设计

概述分布式系统的设计。

第 2 章 为运营而设计

为了实现平稳运营而应该具备的软件功能。

第 3 章 选择服务平台

物理机和虚拟机，私有云和公共云。

第 4 章 应用程序架构

创建 Web 和其他应用程序的基本组件。

第 5 章 伸缩性设计模式

扩展服务所用的基本组件。

附录 A 架构图

2.2.3 自行编写功能	37
2.2.4 与第三方供应商合作	37
2.3 故障模型	38
2.4 小结	38
练习	39
第3章 选择服务平台	40
3.1 服务抽象水平	41
3.1.1 基础设施型服务	41
3.1.2 平台即服务	42

4.3.1 前端	60
4.3.2 应用服务器	61
About the Authors 作者简介	62
4.4 反向代理服务	62
4.5 云规模服务	62
4.5.1 全局负载均衡	63
4.5.2 全局负载均衡方法	63
4.5.3 使用用户特定数据的全局负载均衡	64
4.5.4 内部主干网	64

托马斯 A. 利蒙切利 (Thomas A. Limoncelli) 是具有国际声誉的作家、演说家和系统管理专家。在 Google NYC 的 7 年中, 他曾经担任 Blog Search (博客搜索)、Ganeti 和各种内部企业 IT 服务项目的 SRE。他现于 Stack Exchange 公司任 SRE, 该公司主办 ServerFault.com 和 Stack-Overflow.com 网站。Thomas 的第一份有报酬的系统管理工作是 1987 年在德鲁大学学习时获得的, 此后在小型和大型公司中任职, 包括 AT&T/Lucent 贝尔实验室等。他的著名作品包括《Time Management for System Administrators》(O'Reilly) 和《The Practice of System and Network Administration, Second Edition》(Addison-Wesley)。他的爱好包括草根行动, 为此进行的工作在州乃至国家的层面上得到公认。Thomas 目前住在新泽西州。

斯特拉塔 R. 查卢普 (Strata R. Chalup) 已经领导和管理复杂 IT 项目多年, 所担任的职务包括项目经理和运营主管。Strata 曾经撰写过管理方面的多篇论文, 并且与许多团队合作, 在各种志愿组织 (包括 BayLISA 和 SAGE) 应用其管理技能。她于 1983 年在波士顿的 MIT 开始管理 VAX Ultrix 和 Unisys UNIX, 在硅谷度过了 .com 年代, 为 iPlanet 和 Palm 等客户构建互联网服务。2007 年, 她与 Tom 和 Christina 一起编写了《The Practice of System and Network Administration, Second Edition》。Strata 的爱好包括学习新技术 (如 Arduino 和各种 2D CAD/CAM 设备) 和园艺。她住在加州的圣克拉拉县。

克里斯蒂娜 J. 霍根 (Christina J. Hogan) 在系统管理和网络工程上有 20 年的经验, 足迹遍及美国硅谷、意大利和瑞士。她在小型创业公司、中等规模技术公司和大型跨国公司获得了经验, 多年担任安全顾问, 客户包括 eBay、Silicon Graphics 和 SystemExperts。2005 年, 她和 Tom 因为合著的《The Practice of System and Network Administration》一书而分享了 SAGE 杰出成就奖。她拥有数学学士学位、计算机科学硕士学位、航空工程博士学位和法学文凭。她还曾在某 F1 车队担任空气动力学家 6 年之久, 并代表爱尔兰参加 1988 年国际象棋奥林匹克竞赛, 现居瑞士。

目 录 Contents

译者序	练习	23
前言		
作者简介	第2章 为运营而设计	25
引言	2.1 运营需求	25
	2.1.1 配置	26
	2.1.2 启动和关机	27
	2.1.3 队列排空	28
	2.1.4 软件升级	29
	2.1.5 备份和恢复	29
	2.1.6 冗余性	29
	2.1.7 数据库副本	30
	2.1.8 热切换	31
	2.1.9 单独功能开关	31
	2.1.10 优雅降级	31
	2.1.11 访问控制和速率限制	32
	2.1.12 数据导入控制	33
	2.1.13 监控	33
	2.1.14 审计	33
	2.1.15 调试设施	34
	2.1.16 异常收集	34
	2.1.17 运营文档	35
	2.2 为运营实现设计	35
	2.2.1 从一开始就构建功能	36
	2.2.2 在确定功能时提出请求	36
第一部分 设计：构建系统		
第1章 分布式世界中的设计		8
1.1 大规模的可见性		9
1.2 简单的重要性		10
1.3 构成		10
1.3.1 具有多个后端副本的负载均衡器		10
1.3.2 具有多个后端的服务器		12
1.3.3 服务器树		13
1.4 分布状态		14
1.5 CAP 原则		16
1.5.1 一致性		16
1.5.2 可用性		17
1.5.3 分区可容忍性		17
1.6 松散耦合系统		19
1.7 速度		20
1.8 小结		23

2.2.3 自行编写功能	37	4.3.1 前端	60
2.2.4 与第三方供应商合作	37	4.3.2 应用服务器	61
2.3 改善模型	38	4.3.3 配置选项	62
2.4 小结	38	4.4 反向代理服务	62
练习	39	4.5 云规模服务	62
第3章 选择服务平台	40	4.5.1 全局负载均衡器	63
3.1 服务抽象水平	41	4.5.2 全局负载均衡方法	63
3.1.1 基础设施即服务	41	4.5.3 使用用户特定数据的全局负载 平衡	64
3.1.2 平台即服务	42	4.5.4 内部主干网	64
3.1.3 软件即服务	43	4.6 消息总线架构	66
3.2 机器的类型	44	4.6.1 消息总线设计	67
3.2.1 物理机器	44	4.6.2 消息总线可靠性	67
3.2.2 虚拟机	44	4.6.3 例1: 链接缩短网站	68
3.2.3 容器	46	4.6.4 例2: 员工人力资源数据更新	69
3.3 资源共享水平	48	4.7 面向服务的架构	70
3.3.1 依从性	49	4.7.1 灵活性	70
3.3.2 隐私	49	4.7.2 支持	70
3.3.3 成本	49	4.7.3 最佳实践	71
3.3.4 控制	50	4.8 小结	71
3.4 主机托管	50	练习	72
3.5 选择策略	51	第5章 伸缩性设计模式	73
3.6 小结	53	5.1 总体战略	74
练习	53	5.1.1 识别瓶颈	74
第4章 应用程序架构	54	5.1.2 重新设计组件	74
4.1 单机 Web 服务器	55	5.1.3 计量结果	74
4.2 三层 Web 服务	56	5.1.4 保持主动	75
4.2.1 负载均衡器种类	56	5.2 纵向扩展	75
4.2.2 负载均衡方法	57	5.3 AKF 伸缩立方体	76
4.2.3 共享状态的负载平衡	58	5.3.1 x 轴: 水平复制	76
4.2.4 用户身份标识	59	5.3.2 y 轴: 功能或者服务分割	77
4.2.5 伸缩性	59	5.3.3 z 轴: 面向查找的分割	79
4.3 四层 Web 服务	60	5.3.4 组合	80

5.4	缓存	80	6.6.4	机架	103
5.4.1	缓存效能	80	6.6.5	数据中心	104
5.4.2	缓存布置	81	6.7	超载故障	104
5.4.3	缓存持久性	81	6.7.1	流量浪涌	105
5.4.4	缓存置换算法	82	6.7.2	DoS 和 DDoS 攻击	106
5.4.5	缓存条目失效	82	6.7.3	抓取攻击	106
5.4.6	缓存大小	83	6.8	人为错误	107
5.5	数据分片	84	6.9	小结	108
5.6	线程处理	86	练习		108
5.7	队列	86			
5.7.1	优点	87			
5.7.2	变种	87			
5.8	内容分发网络	87			
5.9	小结	89			
练习		89			
第 6 章 弹性设计模式			第二部分 运营：运行系统		
90			第 7 章 分布式世界中的运营		
6.1	软件弹性胜过硬件可靠性	91	110		
6.2	所有东西最终都会失灵	92	7.1	分布式系统运营	111
6.2.1	分布式系统的 MTBF	92	7.1.1	SRE 和传统企业 IT 的对比	111
6.2.2	传统方法	92	7.1.2	变化和稳定性的对比	112
6.2.3	分布式计算方法	93	7.1.3	SRE 定义	113
6.3	通过备用容量实现弹性	94	7.1.4	大规模运营	114
6.3.1	需要多少备用容量	95	7.2	服务生命周期	116
6.3.2	负载均衡与热备份的对比	95	7.2.1	服务启动	117
6.4	故障域	96	7.2.2	服务退役	120
6.5	软件故障	97	7.3	运营团队组织策略	120
6.5.1	软件崩溃	97	7.3.1	团队成员的工作日类型	122
6.5.2	软件挂起	98	7.3.2	其他策略	124
6.5.3	死查询	98	7.4	虚拟办公室	125
6.6	物理故障	99	7.4.1	沟通机制	125
6.6.1	部件和组件	99	7.4.2	沟通策略	125
6.6.2	机器	101	7.5	小结	125
6.6.3	负载均衡器	102	练习		126
			第 8 章 DevOps 文化		
			128		
			8.1	什么是 DevOps	129
			8.1.1	传统方法	130

8.1.2 DevOps 方法	131
8.2 DevOps 的 3 条道路	131
8.2.1 第一条道路: 工作流	131
8.2.2 第二条道路: 改进反馈	132
8.2.3 第三条道路: 持续试验和 学习	133
8.2.4 小批次更好	133
8.2.5 策略的采用	134
8.3 DevOps 的历史	134
8.3.1 演变	135
8.3.2 网站可靠性工程	135
8.4 DevOps 价值观和原则	136
8.4.1 关系	136
8.4.2 整合	136
8.4.3 自动化	136
8.4.4 持续改进	136
8.4.5 常见的非技术性 DevOps 实践	137
8.4.6 常见的技术性 DevOps 实践	138
8.4.7 DevOps 发行工程实践	139
8.5 向 DevOps 转化	139
8.5.1 准备开始	139
8.5.2 企业层面的 DevOps	140
8.6 敏捷和持续交付	141
8.6.1 什么是敏捷	141
8.6.2 什么是持续交付	141
8.7 小结	143
练习	144

第 9 章 服务交付: 构建阶段 145

9.1 服务交付策略	146
9.1.1 模式: 现代化的 DevOps 方法论	146
9.1.2 反模式: 瀑布方法论	148
9.2 高质量的良性循环	148

9.3 构建阶段的步骤	150
9.3.1 开发	150
9.3.2 提交	150
9.3.3 构建	151
9.3.4 打包	152
9.3.5 注册	152
9.4 构建控制台	152
9.5 持续集成	153
9.6 以软件包作为移交接口	154
9.7 小结	155
练习	156

第 10 章 服务交付: 部署阶段 157

10.1 部署阶段的步骤	157
10.1.1 升级	157
10.1.2 安装	158
10.1.3 配置	158
10.2 测试和批准	159
10.2.1 测试	160
10.2.2 批准	161
10.3 运营控制台	161
10.4 基础设施自动化策略	161
10.4.1 准备物理机器	162
10.4.2 准备虚拟机	162
10.4.3 安装 OS 和服务	163
10.5 持续交付	164
10.6 基础设施即代码	164
10.7 其他平台服务	165
10.8 小结	165
练习	166

第 11 章 升级运行中的服务 167

11.1 卸下服务进行升级	167
11.2 滚动升级	168

11.3	“金丝雀”	169	12.6.3	编译型语言	194
11.4	分阶段试运行	170	12.6.4	配置管理语言	194
11.5	按比例分片	171	12.7	软件工程工具和技术	195
11.6	蓝-绿部署	171	12.7.1	问题跟踪系统	196
11.7	功能切换	171	12.7.2	版本控制系统	197
11.8	在线模式更改	174	12.7.3	软件打包	198
11.9	在线代码更改	175	12.7.4	风格指南	198
11.10	持续部署	175	12.7.5	测试驱动开发	199
11.11	处理失败的代码推送	177	12.7.6	代码评审	200
11.12	发行原子性	178	12.7.7	编写刚好足够的代码	201
11.13	小结	179	12.8	多租户系统	201
	练习	180	12.9	小结	202
第 12 章	自动化	181		练习	203
12.1	自动化方法	182	第 13 章	设计文档	204
12.1.1	剩余原则	182	13.1	设计文档概述	204
12.1.2	补偿原则	183	13.1.1	记录更改和依据	205
12.1.3	互补性原则	184	13.1.2	作为过去决策存储库的文档	205
12.1.4	系统管理自动化	185	13.2	设计文档剖析	205
12.1.5	经验教训总结	185	13.3	模板	207
12.2	工具建设与自动化的对比	186	13.4	文档存档	207
12.2.1	示例：汽车制造	186	13.5	审核工作流程	208
12.2.2	示例：机器配置	187	13.5.1	审核人和批准人	208
12.2.3	示例：账户创建	187	13.5.2	获得签字同意	209
12.2.4	工具很好，自动化更好	187	13.6	采用设计文档	209
12.3	自动化的目标	187	13.7	小结	210
12.4	创建自动化系统	190		练习	210
12.4.1	为自动化投入时间	190	第 14 章	随时待命	212
12.4.2	减少“苦活”	191	14.1	设计值班	212
12.4.3	决定自动化的首要任务	191	14.1.1	从 SLA 开始	213
12.5	如何自动化	192	14.1.2	值班人员花名册	213
12.6	语言工具	192	14.1.3	值日	214
12.6.1	Shell 脚本语言	192	14.1.4	值班表设计	215
12.6.2	脚本语言	193			

14.1.5	值班日程表	216
14.1.6	值班频率	217
14.1.7	通知类型	217
14.1.8	下班时间维护协调	219
14.2	当值	219
14.2.1	当班前的职责	219
14.2.2	常规值班职责	219
14.2.3	警报职责	220
14.2.4	观察、确认、决策、行动	221
14.2.5	值班剧本	221
14.2.6	第三方升级	222
14.2.7	班次结束时的职责	223
14.3	两次当值之间	223
14.3.1	长期修复	223
14.3.2	事后剖析	224
14.4	警报的定期审核	226
14.5	收到太多传呼	227
14.6	小结	228
	练习	228

第 15 章 灾难准备 229

15.1	心态	230
15.1.1	反脆弱系统	230
15.1.2	降低风险	231
15.2	个人培训：灾祸之轮	232
15.3	团队培训：应急演练	233
15.3.1	服务测试	234
15.3.2	随机测试	235
15.4	组织培训：游戏日 /DiRT	235
15.4.1	开始	236
15.4.2	扩大范围	237
15.4.3	实施和后勤	237
15.4.4	经历 DiRT 测试	239
15.5	事故指挥系统	242

15.5.1	工作原理：公共安全领域	243
15.5.2	工作原理：IT 运营领域	243
15.5.3	事故行动计划	244
15.5.4	最佳实践	245
15.5.5	ICS 示例	245
15.6	小结	246
	练习	246

第 16 章 监控基础知识 248

16.1	概述	249
16.1.1	使用监控	250
16.1.2	服务管理	250
16.2	监控信息的消费者	250
16.3	监控的内容	252
16.4	留存期	253
16.5	元监控	254
16.6	日志	255
16.6.1	方法	255
16.6.2	时间戳	256
16.7	小结	256
	练习	256

第 17 章 监控架构与实践 258

17.1	传感与计量	259
17.1.1	黑盒与白盒监控	259
17.1.2	直接计量与合成计量	259
17.1.3	速率与能力监控	260
17.1.4	仪表和计数器	260
17.2	收集	261
17.2.1	推送与拉取	262
17.2.2	协议选择	262
17.2.3	服务器组件与代理、轮询器 的对比	263
17.2.4	中心与区域收集器	263

17.3	分析和计算	264
17.4	警报和升级管理器	265
17.4.1	警报、升级和确认	265
17.4.2	静默与抑制	266
17.5	可视化	267
17.5.1	百分位数	268
17.5.2	堆栈排名	269
17.5.3	直方图	269
17.6	存储	270
17.7	配置	271
17.8	小结	271
	练习	272
第 18 章	容量规划	273
18.1	标准容量规划	274
18.1.1	当前使用量	275
18.1.2	正常增长	276
18.1.3	计划增长	276
18.1.4	余量	276
18.1.5	弹性	277
18.1.6	时间表	277
18.2	高级容量规划	278
18.2.1	确定主要资源	278
18.2.2	了解容量限制	278
18.2.3	确定核心驱动力	279
18.2.4	参与度计量	280
18.2.5	分析数据	280
18.2.6	监控关键指标	284
18.2.7	委派容量规划	285
18.3	资源回归	285
18.4	发布新服务	286
18.5	缩短配给时间	287
18.6	小结	288
	练习	288

第 19 章	建立 KPI	290
19.1	什么是 KPI	291
19.2	创建 KPI	292
19.2.1	步骤 1: 想象理想状况	292
19.2.2	步骤 2: 量化与理想的距离	292
19.2.3	步骤 3: 想象行为的变化 方式	293
19.2.4	步骤 4: 修订和选择	293
19.2.5	步骤 5: 部署 KPI	294
19.3	KPI 示例: 机器分配	294
19.3.1	第一遍	295
19.3.2	第二遍	295
19.3.3	评估 KPI	297
19.4	案例研究: 错误预算	297
19.4.1	相互冲突的目标	297
19.4.2	统一的目标	298
19.4.3	所有人得益	298
19.5	小结	299
	练习	299
第 20 章	卓越运营	301
20.1	卓越运营是什么样子的	301
20.2	如何计量卓越的程度	302
20.3	评估方法论	302
20.3.1	运营职责	303
20.3.2	评估级别	304
20.3.3	评估问题和匹配属性	305
20.4	服务评估	306
20.4.1	确定评估的内容	306
20.4.2	评估每个服务	306
20.4.3	比较不同服务的结果	307
20.4.4	根据结果采取行动	308
20.4.5	评估和项目计划的频率	308
20.5	组织评估	308

20.6 提高级别.....	309	附录 B 分布式计算和云的起源及未来.....	335
20.7 开始着手.....	310	附录 C 伸缩性术语和概念.....	352
20.8 小结.....	311	附录 D 模板和示例.....	356
练习.....	311	附录 E 推荐读物.....	360
 第三部分 附录		后记.....	363
附录 A 评估.....	314	参考文献.....	365

本书的目标是帮助你构建和运行最佳的云环境服务。什么是我们希望创建的理想环境？

商业目标

简单地说，理想环境的最终结果是满足商业目标。这听起来有些乏味，但是实际上在全公司注目的焦点上切入，共同实现同一个目标，是十分激励人心的。为此，我们必须理解商业目标，并且由此倒推出应该构建的系统。

满足商业目标意味着了解这些目标，制定计划实现它们并且消除沿途的障碍。定义良好的商业目标是可计量的，这些计量指标可以自动化收集，自动化生成的仪表盘使每个人了解进展，这种透明度能够增强信任。

下面是商业目标的一些例子：

- 通过网站销售我们的产品。
- 在 99.99% 的时间内提供服务。
- 每个月处理 × 百万笔订单，每月增长 10%。
- 每周两次推出新功能。
- 在 24 小时内修复重大缺陷。

在我们的理想环境中，业务和技术团队以可预测和可靠的方式满足他们的目标和项目目标。因此，两类团队都信任其他团队能够满足未来的目标。结果是，团队可以更好地制定计划。他们可以制定更为积极的计划，因为确信外部依赖不会失效。在这种情况下，计划可以更加积极，这种方法会形成向上的螺旋，加速整个公司的进展，惠及每个人。

引言 Introduction

本书的目标是帮助你构建和运行最佳的云规模服务。什么是我们希望创建的理想环境？

商业目标

简单地说，理想环境的最终结果是满足商业目标。这听起来有些乏味，但是实际上在全公司注目的焦点上切入，共同实现同一个目标，是十分激动人心的。为此，我们必须理解商业目标，并且由此倒推出应该构建的系统。

满足商业目标意味着了解这些目标、制定计划实现它们并且消除沿途的障碍。定义良好的商业目标是可计量的，这些计量指标可以自动化收集。自动化生成的仪表盘使每个人了解进展，这种透明度能够增强信任。

下面是商业目标的一些例子：

- ❑ 通过网站销售我们的产品。
- ❑ 在 99.99% 的时间内提供服务。
- ❑ 每个月处理 x 百万笔订单，每月增长 10%。
- ❑ 每周两次推出新功能。
- ❑ 在 24 小时内修复重大缺陷。

在我们的理想环境中，业务和技术团队以可预测和可靠的方式满足他们的目标和项目目标。因此，两类团队都信任其他团队能够满足未来的目标。结果是，团队可以更好地制定计划。他们可以制定更为积极的计划，因为确信外部依赖不会失效。在这种情况下，计划可以更加积极，这种方法会形成向上的螺旋，加速整个公司的进展，惠及每个人。

理想系统架构

理想的服务是在稳定的架构上构建的，它能够满足当前的服务需求，并且在系统变得较为流行、接收更多流量时提供明显的成长路径。这种系统对故障具备弹性，架构不会对故障感到意外，将其作为异常情况对待，而是接受硬件和软件故障作为信息技术（IT）物性的一部分。结果是，这种架构包含应对故障的冗余和弹性特征，组件失效时系统仍可存活。

组成服务的每个子系统本身也是服务。所有子系统都可以通过应用程序编程接口（API）编程，因此，整个系统是互联子服务的生态系统，这称作面向服务架构（SOA）。因为这些系统都使用相同的底层协议通信，具有管理上的一致性。每个子服务相互松散耦合，因此这些服务可以独立伸缩、升级或者替换。

基础设施的几何结构采用电子方式描述，这种电子描述由 IT 自动化系统阅读，然后在没有人为干预的情况下构建生产环境。基于这种自动化，整个基础设施可以在其他地方重建，软件工程师利用自动化建立环境的微版本，供个人使用。质量和测试工程师利用自动化创建用于系统测试的环境。

无论我们是使用物理机器还是虚拟机器，也不管是使用我们运行的还是由云提供商托管的数据中心，都能实现这种“基础架构即代码”。对于虚拟机，显然有 API 可以用于准备新机器。但是，即使使用物理机器，从裸机到工作系统的整个流程也可以自动化。在我们的理想世界中，自动化使组合物理和虚拟机器创建环境成为可能。开发人员可以从虚拟机构建环境，生产环境可能包含物理和虚拟机的组合。额外容量的临时和意外需求可能需要在一段时期内将生产环境扩展到一个或者多个云提供商。

理想的发行过程

理想的环境具备从代码开发到运营的平稳流程。传统上（不是在我们的理想环境中）顺序是这样的：

- 1) 开发者在存储库中登记代码。
- 2) 测试工程师将代码投入一系列测试。
- 3) 如果所有测试通过，发行工程师将构建一个用于部署软件的“包”。大部分文件来自于源代码库，但是有些文件可能必须来自其他来源，如图形部门或者文档编写者。
- 4) 创建一个测试环境；如果没有采用“基础架构即代码”模式，这可能需要花费数周。
- 5) 软件包被部署到测试环境中。
- 6) 测试工程师进一步执行测试，重点是子系统之间的交互。
- 7) 如果所有测试成功，则将代码投入生产环境。
- 8) 系统管理员升级系统同时寻找缺陷。
- 9) 如果有缺陷，则将软件回滚。

人工进行上述步骤会招致许多风险，这是因为事先得具备合适的人员、上述步骤每次都以相同的方式完成、没有人会犯错误且所有任务都会及时完成等假设。

错误、程序缺陷和偏差当然会发生——因此缺陷会被传递到下一个阶段。发现错误时，工作流倒转，负责前一阶段的团队成员被告知需要修复他们的问题。这意味着工作无法取得进展、损失了时间。

对某种高风险过程的典型反应是尽可能少做。因此就产生了尽可能少发行的念头，结果是“重大发行版本”每年只投放一两次。

但是，将许多更改一次性批量进行，实际上造成了更大的风险。如何确保同时发布的几千项更改在第一次尝试中全部取得成功？我们做不到。因此，我们变得更加不情愿和恐惧变化。很快，变化几乎不可能发生，创新的脚步也停止下来。

在我们的理想环境中不是这样的。

在理想环境中，我们找到自动化的方法，消除软件构件、测试、发行和部署过程中的所有人工步骤。自动化精确、一致地执行测试，避免缺陷传递到下一步骤。结果是，流程只有一个方向：向前。

我们的理想环境并不创建“重大发行版本”，而是创建“微发行版本”。通过进行多次部署，每次进行少量小规模更改，降低了风险。实际上，我们每天可以进行100次部署。

1) 当开发者登记代码时，有一个系统检测到这一事实，触发一系列自动化测试，这些测试验证代码的功能。

2) 如果这些测试通过，则开始构建软件包的过程，并且以完全自动化的方式运行。

3) 新软件包的成功创建触发测试环境的创建。构建测试环境过去需要漫长的一周来连接电缆和安装机器，但是利用“基础设施即代码”模式，整个环境可以在没有人工干预的情况下快速创建。

4) 测试环境完成时，运行一系列自动化测试。

5) 成功完成之后，新软件包投入生产环境试运行。试运行也是自动化的，但是有序且审慎。

6) 某些系统首先升级并监视其故障。由于测试环境构建时使用与生产环境一样的自动化手段，两者之间的差别应该非常小。

7) 如果没有发现故障，新的软件包向越来越多的系统铺开，直到整个生产环境都被升级。

在我们的理想环境中，所有问题都在投产之前捕捉到。也就是说，试运行并不是测试的一种形式，在此期间发生的故障基本上都被排除。但是，如果确实发生故障，应该将其视为严重的问题，有理由停止新发行版本投产，直到根源分析完成。增加测试以检测和预防未来发生同一故障，从而使系统随着时间推移变得更加坚固。

因为采用上述的自动化手段，发行工程、质量保证及部署的传统任务实际上不同于传统公司。许多个小时的人工被节约下来，为改善打包系统、改善软件质量和调整部署过程

留出了更多时间。换言之，人们将更多的时间花在已完成工作的改进上，而不是工作本身。

对第三方软件也采用类似的过程。并不是所有系统都是自产或者提供源代码的。部署第三方服务和产品遵循类似的发行、测试、部署模式。但是，因为这些产品和服务是在外部开发的，它们需要的过程略有不同。发行新版本的频率可能较低，我们对每个新发行版本的控制也较弱。这些组件需要的测试通常与功能、兼容性和集成有关。

理想的运营

一旦代码投产，运营目标就成为优先考虑的因素。软件被“仪表化”以便进行监控，收集处理来自外部用户和内部 API 事务所花费的时间的相关数据，还要监控内存使用率等指标。收集这些指标以便根据数据做出运营决策，而非依靠猜测、运气或者希望。这些数据需要存储许多年，以便用于预测未来的容量需求。

使用计量可以在内部问题还比较小、远不会造成用户可见中断的时候发现它们，我们可以在问题爆发之前修复。真正的停机事故很少见，在出现时进行深入调查。当发现问题时，有某种过程确保它们的识别、处理和快速解决。

自动化系统检测问题并警告所有值班人员。我们采用轮流值班制度，使每个班次接收的警报次数可控。在任何时候都有一位主要值班人员，他最先接收到警报，如果这个人没有及时响应，则向第二个人发出警报。值班安排提前准备充分，以便人员计划假期、娱乐活动和个人时间。

对于如何处理各种可能产生的警报都有“剧本”。各种警报都有文档，从技术上描述出现的错误、对业务的影响和修复问题的方法。这些“剧本”持续改进，任何值班人员都使用“剧本”修复问题，如果证明“剧本”不充足，将有精心定义的升级路径，通常导向相关子系统的值班人员。开发人员也是值班轮次的参与者，这样，他们就能够了解所构建系统在运营上的“痛点”。

所有故障都有相应的对策，这些对策或人工激活，或自动激活。频繁激活的对策始终是自动化的。我们的监控系统检测过度使用，因为这代表着较大的问题。监控系统收集工程师使用的内部指标数据，减少故障率、改进对策。

对策激活的次数越少，我们就越不确信它在下一次需要的时候是否有效。因此，不经常激活的对策必须通过有意造成故障而定期、自动演练。正如我们要求学生进行消防演习，使每个人知道在紧急情况下怎么做一样，对于运营实践也要进行演练。这样，我们的团队在实施对策时就变得很有经验，并对工作表现出自信。如果数据库故障切换过程因为意外的依赖性而失效，最好在周一上午十点的实操演习中研究，而不是在周日凌晨4点的停机事故中研究。

我们通过反复演练来减少风险，而不是回避问题。通过重复改善某些环节的技术术语是“实践”。我们坚信，实践造就完美。

我们的理想环境是可以自动伸缩的。需要更多容量时，附加的容量来自于内部或者外部云提供者。当重新架构的解决方案比简单地分配更多 RAM、磁盘或者 CPU 更好时，我们的仪表盘将会指明。

系统的收缩也是自动化的。当系统过载或者降级时，我们不再用“503—Service Unavailable”错误拒绝用户，相反，系统会自动切换到使用更少资源的算法。带宽完全被占用了？启动低带宽版本的服务，显示较少图形或者简化的用户界面。数据库损坏了？只读版本的服务能够满足大部分用户的要求。

服务的每个功能都可以单独启用或者禁用。如果某个功能造成负面的结果，如安全漏洞或者意外地劣化性能，可以禁用它而无须部署不同的软件发行版本。

当某项功能修订时，新代码不会淘汰旧功能。可以禁用新行为，展示旧行为。这在新用户界面投入试运行特别有用。如果某个发行版本既能生成旧的用户界面，又能生成新的界面，这样每个用户都可以单独禁用某个界面。这使我们可以从“早期访问”用户那里得到反馈，在正式发行日，成功地为越来越大的用户群体启用新功能。如果发现性能问题，该功能很容易恢复或者完全关闭。

在理想的环境中有卓越运营保健法。就像刷牙一样，我们定期进行保持运营健康的活动。我们为如何处理每个对策、过程和警报维护清晰的最新文档。对过于频繁出现的警报进行微调而不忽略。将未解决缺陷的数量保持在最低限度。停机事故之后会发布事后剖析报告，提出未来系统改进的建议。任何“快速修复”之后都进行根源分析并实施长期修复。

最重要的是，开发人员和运营人员不将自己视为两个不同的团队。他们只是更大团队中的不同专业。有些人写的代码比其他人多；有些人负责的运营项目比其他人多。所有人的共同责任是保持高的正常运行时间。为此，所有成员轮流值班（接听电话和传呼）。开发人员在感受到运营的痛苦时，也会激发其改善影响运营的代码的积极性。如果运营人员想要有建设性地与开发人员协作，就必须理解开发过程。

现在，你已经知道了我们对理想环境的愿景，本书剩下的部分将解释创建和运营这种环境的方法。

■ 第1章 分布式世界中的设计

■ 第2章 分布式系统设计

■ 第3章 各种服务平台

■ 第4章 应用程序架构

■ 第5章 微服务设计模式

■ 第6章 云原生设计模式

QPS：每秒查询次数。通常是每秒接收的点击数或者API调用次数。

流量：查询、API调用或者其他发送给服务器的请求的统称。

高性能 (Performant)：系统的性能符合（达到或者超过）设计要求。这个新闻是“性能” (Conformant) 组合而成的。

管理服务器与其他机器通信方式的协议

第1章

第一部分 Part 1

设计：构建系统

1.1 大规模的可见性

为了管理大规模分布式系统，必须拥有系统的可见性。检查内部状态的能力——操作内省 (Introspection)——是运营、调试、调整和维修大型系统所必需的。

随着系统规模的增加，系统变得越来越复杂。系统管理员需要能够实时监控系统的运行状态，并能够在出现问题时快速定位和解决。为了实现这一点，系统需要具备自我监控和自我报告的能力。这通常通过日志记录、性能指标收集和异常检测来实现。系统管理员可以利用这些工具来了解系统的健康状况，并在必要时采取行动。

此外，分布式系统还需要具备可扩展性。这意味着系统能够随着负载的增加而自动调整资源，以保持性能和可用性。这通常通过负载均衡、自动扩缩容和分布式存储来实现。

- 第1章 分布式世界中的设计
- 第2章 为运营而设计
- 第3章 选择服务平台
- 第4章 应用程序架构
- 第5章 伸缩性设计模式
- 第6章 弹性设计模式

此外，分布式系统还需要具备高可用性。这意味着系统能够在部分组件发生故障时继续运行，并且能够快速恢复。这通常通过冗余、故障转移和容错设计来实现。系统管理员可以利用这些工具来了解系统的健康状况，并在必要时采取行动。

分布式世界中的设计

构造软件设置有两种方法：一种是使其简单化，明显没有任何缺陷；另一种则是使其复杂化，没有明显的缺陷。

——C. A. R. Hoare

1980 年 ACM 图灵奖获奖演讲

Google 搜索是如何工作的？Facebook 动态时报（Timeline）是如何不断更新的？Amazon 是如何扫描不断增长的商品目录，告诉你购买这个商品的人还买了袜子的？

这是魔术吗？不，这就是分布式计算。

本章概述设计使用分布式计算技术的服务所需涉及的环节。这些技术是所有大型网站实现其规模、伸缩性、速度和可靠性时使用的。

分布式计算是构建大型系统，将工作分配到许多台机器的艺术。相比之下，传统计算系统中由单台计算机运行提供服务的软件，而客户端 / 服务器计算则由许多台机器远程访问一个集中化服务。在分布式计算中，通常有成百上千台机器一起工作，提供大规模服务。

分布式计算在许多方面不同于传统计算。大部分不同之处是缘于系统本身的规模。分布式计算可能涉及数百甚至数千台计算机，为几百万用户提供服务，处理数十亿（有时甚至达到数千亿）次查询。

必知术语

服务器：提供功能或者应用程序接口（API）的软件（不是一个硬件设备）。

服务：由许多服务器组成的用户可见系统或者产品。

机器：虚拟或者物理机器。

QPS: 每秒查询次数。通常是每秒接收的点击数量或者 API 调用次数。

流量: 查询、API 调用或者其他发送给服务器的请求的统称。

高性能 (Performant): 系统的性能符合 (达到或者超过) 设计要求。这个新词是“性能”(Performance) 和“符合”(Conformant) 组合而成的。

应用编程接口 (API): 管理服务器与其他机器通信方式的协议。

速度很重要。对于一个服务来说, 快速、反应灵敏是竞争优势。如果不能在 200 ms 以内响应, 用户会认为网站迟钝。网络延时占据了这段时间的大部分, 留给服务构建网页的时间很少。

在分布式系统中, 故障是很正常的。硬件故障很少见, 但是由于有数千台机器, 故障就变得常见了。因此, 必须假定故障的存在, 设计变通方法和预测故障的软件。故障是预期的一部分。

由于分布式系统的规模很大, 运营必须实现自动化。人工完成涉及数百或者数千台机器的任务是不可思议的。自动化对于软件的准备和部署、日常运营和故障处理都至关重要。

1.1 大规模的可见性

为了管理大型分布式系统, 必须拥有系统的可见性。检查内部状态的能力——称作内省 (Introspection) ——是运营、调试、调整 and 维修大型系统所必需的。

在传统系统中, 可以想象由一位对系统有足够认识的工程师密切注视关键组件或者根据经验“就能知道”哪里出了问题。在大型系统中, 这种水平的可见性必须通过设计提取信息并使其可见的系统主动地创建。没有任何个人或者团队能够人工监控所有部件。

因此, 分布式系统的组件必须能够产生足以描述系统中所发生情况的丰富日志。然后, 这些日志汇集到一个中心位置, 供收集、存储和分析。系统可能记录很高级别的日志, 如在用户每次购物、Web 查询或者 API 调用时记录。也可能记录低级别信息, 如重要代码中每次函数调用的参数。

系统应该输出指标, 它们应该统计感兴趣的事件, 如特定 API 被调用多少次, 并且提供计数器的访问手段。

在许多情况下, 可以使用特殊 URL 查看这些内部状态。例如, Apache HTTP Web 服务器有一个“服务器状态”(Server-status) 页面 (<http://www.example.com/server-status/>)。

此外, 分布式系统的组件往往评价自身健康状况并使这些信息可见。例如, 某个组件可能有一个 URL, 可以输出系统是否已为接收新请求做好了准备 (OK)。在接收输出不是字节“O”和后续的字节“K”(包括完全没有响应) 时, 说明系统不打算接收新请求。负载均衡器使用这一信息确定服务器是否健康, 是否准备接收流量。当服务器正在启动和仍在初始化、正在关闭且不再接收新请求但正在处理已提交请求时, 会发送负面的响应。

1.2 简单的重要性

设计在能够满足服务需求的同时尽可能保持简单是很重要的。随着时间的推移，系统成长且变得更加复杂。从已经很复杂的系统入手，意味着一开始就处于不利地位。

出色的运营工作需要人们掌握系统的心智模型。在工作时，我们设想系统的运行模式，并使用这个心智模型跟踪系统的工作方式，在系统不正常时进行调试。系统越复杂，掌握精确的心智模型就越困难。过度复杂的系统会造成这样一种情况：任何人在任何时候都无法完全理解系统。

在《The Elements of Programming Style》中，Kernighan 和 Plauger（1978）写道：

调试的难度两倍于代码的编写。因此，如果你尽可能巧妙地编写代码，那么调试代码也就不需要那么多的技巧。

对于分布式系统也是如此，为简化设计所花费的每一分钟，在运营的时候会一再得到回报。

1.3 构成

分布式系统由许多较小的系统组成。在本节中，我们详细研究 3 种基本构成模式：

- 具有多个后端副本的负载均衡器。
- 具有多个后端的服务器。
- 服务器树。

1.3.1 具有多个后端副本的负载均衡器

第一种构成模式是具有多个后端副本的负载均衡器。如图 1.1 所示，请求发送给负载均衡服务器。对于每个请求，服务器选择一个后端（Backend）并将请求转发给它。响应返回给负载均衡服务器，由此转发给原始请求者。

后端被称为副本（Replica）是因为它们互为对方的复制品。请求发给任何一个副本都应该产生相同的响应。

负载均衡器必须始终知道哪些后端存活、正做好准备接收请求。负载均衡器向后端每秒发送数十次健康检查（health check）查询，如果健康检查失败，则停止向该后端发送请求。健康检查是简单的查询，应该快速执行并返回系统是否应该接收流量的结果。

选择向哪个后端发送请求可能很简单，也可

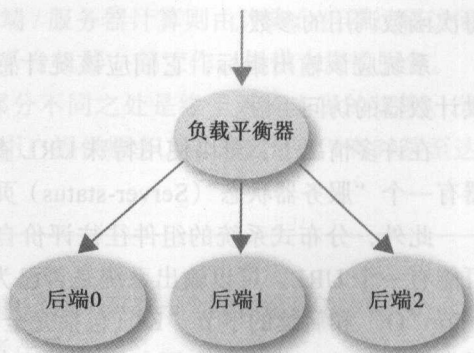


图 1.1 具有许多副本的负载均衡器

能很复杂。简单的方法通常以循环方式交替使用后端——这种方法被称为**循环法**（Round-robin）。但是，有些后端可能比其他后端更强大，因此在使用比例循环方案时更常被选择。更复杂的解决方案包括**最小负载**（least loaded）方案。在这种方法中，负载平衡器跟踪每个后端的负载，始终选择负载最小的后端。

选择最小负载的后端似乎很合理，但是简单地实施可能成为一场灾难。后端可能在真正过载很久之后才显示负载的信号。这个问题的发生是因为难以准确地计量系统的负载，如果负载以最近与该服务器建立连接的数量计量，这一定义就忽视了有些连接持续很长时间，而其他连接可能很快断开这一事实。如果计量基于 CPU 利用率，那么又忽视了输入/输出（I/O）过载。经常使用的计量是最后 5 分钟的平均负载。这种计量有一个问题：作为平均值，它们反映的是过去而非现在。因此，负载的急剧和突然增加在一段时间内不会反映到平均值上。

想象一个具有 10 个后端的负载平衡器，每个运行于 80% 的负载。此时加入一个新后端，因为它是新加入的，所以没有任何负载，从而成为最小负载后端。简单最小负载算法将把所有流量发送到这个新后端；其他 10 个后端没有接收任何流量。很快，这个新后端就会陷入“泥沼”。没有一个后端能够处理之前由 10 个后端处理的流量，使用最近平均值意味着旧的后端将在几分钟内发出高负载的虚假报告，而新的后端仍然虚假地报告低负载。

使用这种方案，负载平衡器在一段时间内将会认为新机器的负载低于其他的所有机器。在这种情况下，新机器可能过载以致崩溃并重启，或者被试图纠正这一状态的系统管理员重启。当它重新投入服务时，又将重复上述循环。

这种情形使循环方法看上去很不错。稍复杂一些的最小负载方案采用某种控制，永远不会连续向同一机器发送超过某一数量的请求。这称作**慢启动**（slow start）算法。

简单最小负载算法引发的麻烦

如果不使用慢启动，负载平衡器会造成许多问题。一个著名的例子发生在 2001 年 9 月 11 日恐怖袭击当天的 CNN.com 网站。许多民众试图访问 CNN.com 导致后端超载，其中一个后端崩溃，在恢复之后因为简单最小负载算法将所有流量发送给它而再度崩溃。在它停机之后，其他后端也超载并崩溃。所有后端逐一超载、崩溃，然后接收所有流量而再次崩溃。

结果是，当系统管理员赶来了解情况时，该服务实际上已经无法使用。在他们进行防御时 Web 还是新生事物，没有人有处理类似 9 月 11 日遇到的突发高流量的经验。

CNN 采用的解决方案是停止所有后端并同时启动它们，使它们全部显示零负载、接收相同的流量。

CNN 团队后来发现，几天以前，他们的负载平衡器收到了一个软件升级，但是尚未安装，这个升级中加入了慢启动机制。

1.3.2 具有多个后端的服务器

第二种构成模式是具有多个后端的服务器。服务器接收请求，向许多后端服务器发送查询，合并所有应答组成最后的响应。这种方法通常用在原始查询很容易分解为一些独立查询，并且这些查询可以组合形成最终应答的情况。

图 1.2a 展示了简单的搜索引擎如何在多个后端的帮助下处理查询。前端接收请求，并将请求转发给许多后端服务器。拼写检查器发出应答信息，使搜索引擎能够建议替代的拼写。Web 和图像搜索后端以与查询相关的网站和图像列表作为应答。广告服务器以与查询相关的广告应答。接收到应答之后，前端使用这些信息构造组成用户搜索结果页面的 HTML，然后作为应答发送。

图 1.2b 展示了具有复制、负载均衡和后端的同一架构，使用相同原则，但是系统可以更好地伸缩并在故障时存活。



图 1.2 这个服务由一个服务器和许多后端组成

这种构成有许多优势。后端并行完成自己的工作。不必等待一个后端处理结束，就可以开始处理下一个应答。系统的耦合很松散，一个后端失效时，页面仍然可以通过填入默认信息或者将该区域留空来构造。

这种模式还能够进行某种相对复杂的延迟管理。假定这个系统预期在 200 ms 或者更短的时间内返回结果。如果一个后端因为某种原因而处理缓慢，前端不用等待它。如果它花费 10 ms 就可以构成并发送结果 HTML，在 190 ms 时，前端可以放弃缓慢的后端，用所具有的信息生成页面。这种管理延迟时间预算的能力非常强大，例如，如果广告系统缓慢，搜索结果可以不显示任何广告。

更清楚一点说，“前端”和“后端”这两个术语是视角的问题。前端向后端发送请求，后端以某种结果应答。服务器既可能是前端也可能是后端。在前一个例子中，服务器对 Web 浏览器来说是后端，对拼写检查服务器来说是前端。

这一模式有许多变种。每个后端可以复制，以增加容量或者弹性。缓存也可以在多个级别上完成。

扇出（fan out）这一术语指的是一个查询造成许多新查询，每个查询对应一个后端的事实。查询“扇出”到单独的后端，而应答按照设置“扇入”到前端，组合为最终结果。

任何扇入的情况都有拥塞的危险。小的查询往往可能造成大的响应，因此扇出时使用的少量带宽往往不足以承受扇入。这可能造成网络连接拥塞和服务器过载。如果查询和响应是一致的，或者偶然有大规模响应，那么很容易设计具有合适网络数量和服务器容量的系统。困难的是当出现突然、不可预测的大规模应答爆发的时候，有些网络设备专门设计为在突发流量时动态供给更多缓冲区空间，以处理这种状况。

同样，后端也可以限制自身的速率，避免一开始就造成上述情况。最后，前端可以通知后端减速，控制发出的新查询数，或者实施紧急措施更好地处理洪泛，管理自身的拥塞。最后一种选项在第5章中讨论。

1.3.3 服务器树

另一种基本构成模式是服务器树（Server Tree），正如图1.3所示，在这种方案中，许多服务器相互协作，其中之一作为树根，下方是父服务器，树的底部是叶子服务器（在计算机科学中，树是上下颠倒的）。这种模式一般用于访问大的数据集（语料库）。语料库很大，任何机器都无法容纳，因此每个叶子服务器存储整个数据集的一部分（分片）。

为了查询整个数据集，根服务器接收原始查询并转发给父服务器，父服务器将查询转发给叶子服务器，由其搜索语料库中的各个部分。每个叶子服务器将其发现发送给父服务器，父服务器排序和过滤结果并将其转发给根服务器。根服务器取得所有父服务器的响应，合并结果并以完整的答案应答。

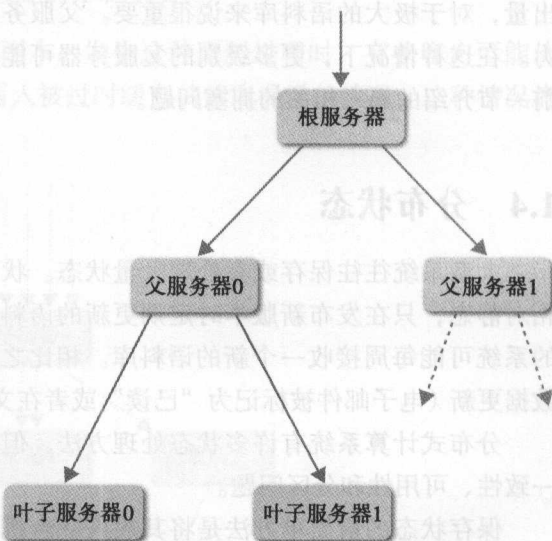


图 1.3 服务器树

想象一下，你想要找出乔治·华盛顿在一本百科全书中被提及几次，可以按照顺序读取每一卷得到答案，也可以将每一卷交给不同的人，让他们并行搜索自己手中的书。后一种方法会更快完成任务。

这种模式的主要好处是允许并行搜索大规模语料库，不仅叶子服务器可以并行搜索自己的语料库份额，父服务器执行的排序和分类也是并行的。

例如，想象一个由美国国会图书馆中所有书籍正文组成的语料库。一台计算机无法容纳这样的库，所以信息被分布给几十万台叶子机器。除了叶子机器之外，还有父服务器和根服务器。搜索查询将发往根服务器，根服务器依次将查询转发给所有父服务器。每台父服务器向所有下属叶节点重复发出查询。一旦叶子结点应答，父服务器按照相关度分类排

序结果。

例如，一台叶子机器可能以存在于某本书同一段内的所有查询单词应答，而另一本书只存在其中一些单词（相关度较低），或者这些单词都存在，但是不在同一段或同一页（相关度更低）。如果查询最好的 50 个答案，父服务器将向根服务器发送前 50 个结果，抛弃其余结果。然后，根服务器接收每个父服务器的结果，选择其中最好的 50 个构造应答。

这种方案也允许开发者工作于预算的延时之内。如果快速回答比完美的答案更重要，父服务器和根服务器在临近最大允许延迟时不必等待慢速的应答。

这种模式可能有许多变种。可能存在冗余服务器，采用负载均衡方案将工作分布到这些服务器中，并绕开失效的服务器。增加叶子服务器可以使搜索的语料库部分更小，语料库的每个分片也可以放置在多个叶子服务器上，增强可用性。增加每级父服务器的数量可以增加排序分类结果的能力。更多级别的父服务器使树变得更高，这样可以得到更大的扇出量，对于极大的语料库来说很重要。父服务器可能提供缓存功能，减小叶子服务器的压力。在这种情况下，更多级别的父服务器可能提高缓存的效能。这些技术也有助于缓解与前一节介绍的扇入相关的拥塞问题。

1.4 分布状态

大型系统往往保存或者处理大量状态。状态由频繁更新的数据组成，如数据库。这与相对静态，只在发布新版本时定期更新的语料库形成了对比。例如，搜索美国国会图书馆的系统可能每周接收一个新的语料库。相比之下，电子邮件系统随着数据不断到达、当前数据更新（电子邮件被标记为“已读”或者在文件夹之间移动）、数据删除而不断变动。

分布式计算系统有许多状态处理方法。但是，它们都涉及某种复制和分片，这带来了一致性、可用性和分区问题。

保存状态的最简单方法是将其放入一台机器，如图 1.4 所示。遗憾的是，这种方法很快就会达到极限：单台机器只能保存有限的状态，如果一台机器停机，我们就无法访问所有状态。机器的处理能力有限，这意味着它所能处理的并发读写数量也有限。

在分布式计算中，我们通过在单独的机器中存储分片来存储状态，可以存储的状态数量仅受所能获得的机器数量限制。此外，每个分片保存在多台机器上；因此，单台机器故障不会影响任何状态的访问。每个副本每秒可以处理一定数量的查询，所以我们可以增加副本数量，设计处理任意并发读写请求数量的系统。这种系统如图 1.5 所示，图中每秒接收分布到 3 个分片上的 N 个查询（ N qps），每个请求都被复制到 3 条路径上。结果是，每个特定副本服务器平均接收所有查询的 $1/9$ 。

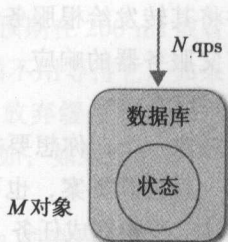


图 1.4 状态保存在一个位置；没有采用分布式计算

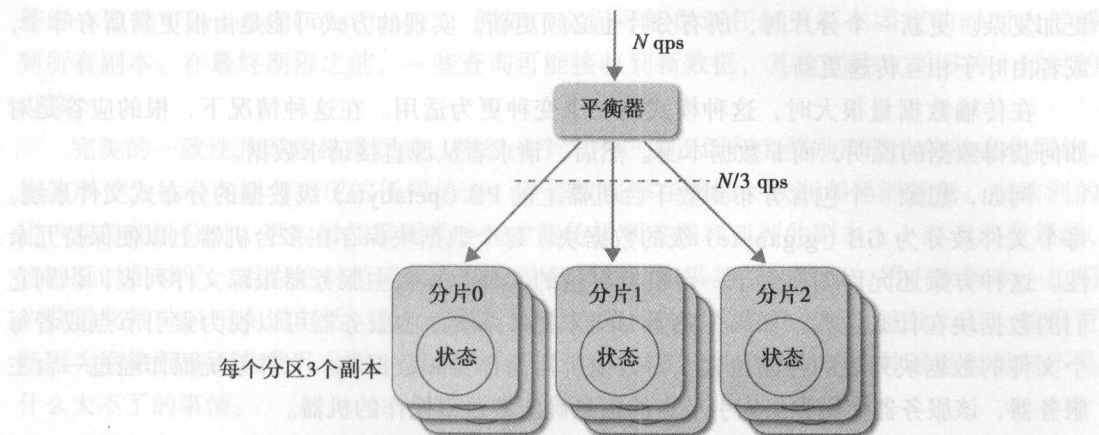


图 1.5 分片、复制的分布式状态

写入或者更新状态请求需要更新所有副本。发生这种更新过程时，有些客户可能从尚未更新的陈旧副本中读取。图 1.6 展示了写入被过时缓存的读取混淆的情况，这种情况将在下一小节中讨论。

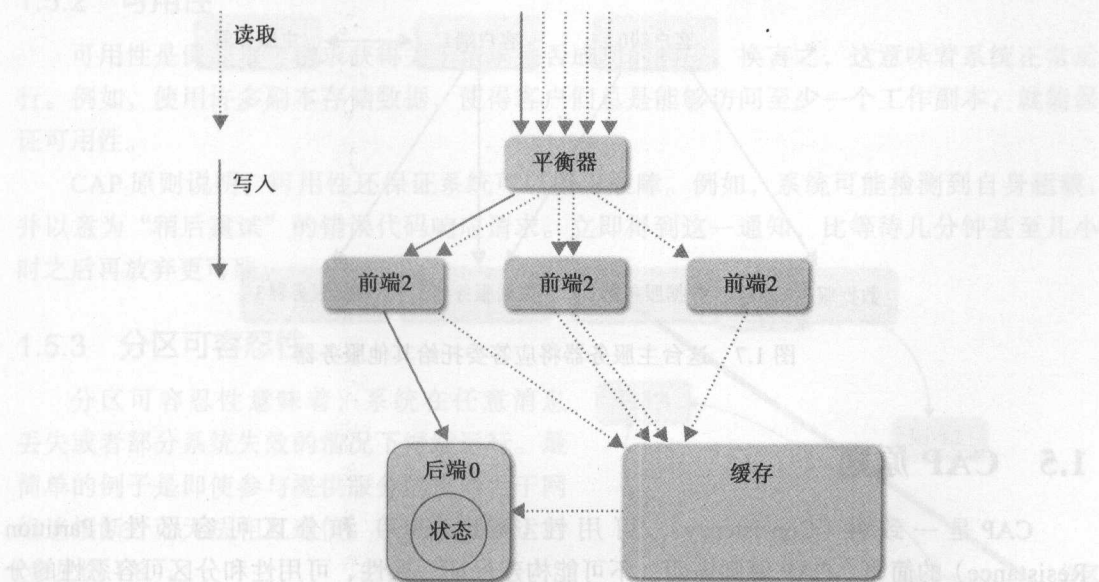


图 1.6 使用缓存数据更新状态造成不一致的视图

在最简单的模式中，根服务器接收存储或者检索状态的请求。它确定包含这部分状态的分片，并将请求转发给对应的叶子服务器。然后，应答沿树上行，这与前一小节描述的服务器树模式类似，但是有两个不同之处。首先，查询转发到单个叶子而不是所有叶子。其次，请求可能是更新（写入）请求，而不只是读请求。当分片保存在许多副本上时，更新

更加复杂。更新一个分片时，所有分片也必须更新。实现的方式可能是由根更新所有叶子，或者由叶子相互传递更新。

在传输数据量很大时，这种模式的一个变种更为适用。在这种情况下，根的应答是对如何获得数据的说明，而非数据本身。然后，请求者从源直接请求数据。

例如，想象一个包含分布到数千台机器上的 PB (petabyte) 级数据的分布式文件系统。每个文件被分为 GB (gigabyte) 级的数据块。每个数据块保存在多台机器上以便保持冗余性。这种方案还允许创建大于一台机器容量的文件。一台主服务器跟踪文件列表，识别它们的数据块在什么位置。如果你熟悉 UNIX 文件系统，主服务器可以视为索引节点或者每个文件的数据块列表的存储位置，而其他机器保存实际数据块。文件系统操作经过一台主服务器，该服务器使用类似索引节点的信息确定涉及该操作的机器。

想象大的读请求进入的情况，主服务器确定文件中有几 TB (terabyte) 保存在某台机器上，另外几 TB 保存在另一台机器上。它可以向两台机器上请求数据，并将其转发给发出请求的系统，但是主服务器很快会在接收和转发巨大的数据块时超载。作为替代，它以包含数据的机器列表作为应答，而请求者直接联系这些机器获得数据。这样，主服务器就不用作为大规模数据传输的中介，如图 1.7 所示。

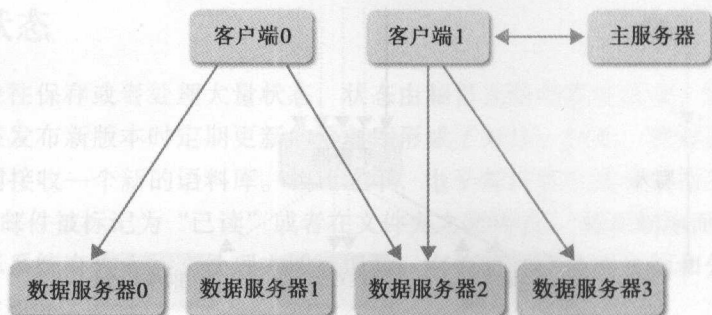


图 1.7 这台主服务器将应答委托给其他服务器

1.5 CAP 原则

CAP 是一致性 (Consistency)、可用性 (Availability) 和分区可容忍性 (Partition Resistance) 的简写。CAP 原则声明，不可能构建保证一致性、可用性和分区可容忍性的分布式系统，三者中的一个或者两个可能实现，但是不可能同时实现全部条件。在使用这种系统时，你必须知道哪些条件得到保证。

1.5.1 一致性

一致性意味着所有节点在同一时刻看到相同的数据。如果有多个副本且正在处理更新，即使从不同的副本读取，所有用户也应该同时看到更新。不能保证一致性的系统可能提供

最终一致性 (Eventual Consistency)。例如, 它们可能保证任何更新在固定的时间内传播到所有副本。在最终期限之前, 一些查询可能接收到新数据, 其他则接收到旧的、过时的应答。

完美的一致性并不总是重要的。想象一个为用户的正面行为奖励声誉分的社会化网络。你的声誉分显示在姓名出现的任何地方。声誉数据库在美国、欧洲和亚洲复制。在欧洲的用户获得奖励分时, 更改可能需要几分钟才能传播到美国和亚洲的副本上。对于此类系统, 这种延时足够了, 因为绝对精确的声誉分没有必要。如果美国和亚洲的用户在其中一位用户获得奖励分时正在通电话, 另一位用户将会在几秒钟之后看到更新, 这就行了。如果更新因为网络拥塞而花费几分钟, 或者因为网络故障而花费几个小时, 这种延迟也仍然不是什么大不了的事情。

现在想象在这种系统上构建的银行应用。在美国的一位用户和在欧洲的另一位用户可能同时从同一个账户上取款。每个人使用的 ATM 将查询最近的数据库副本, 该副本声称有足够的款项, 可以取款。如果更新的传播很慢, 两个人都会在银行意识到钱已经被取走之前获得现金。^①

1.5.2 可用性

可用性是保证每个请求获得关于请求是否成功的响应。换言之, 这意味着系统正常运行。例如, 使用许多副本存储数据, 使得客户们总是能够访问至少一个工作副本, 就能保证可用性。

CAP 原则说明, 可用性还保证系统可以报告故障。例如, 系统可能检测到自身超载, 并以意为“稍后重试”的错误代码响应请求。立即得到这一通知, 比等待几分钟甚至几小时之后再放弃更可取。

1.5.3 分区可容忍性

分区可容忍性意味着, 系统在任意消息丢失或者部分系统失效的情况下继续运行。最简单的例子是即使参与提供服务的机器由于网络连接断开而无法相互通信, 系统仍继续运行 (参见图 1.8)。

回到我们的副本示例, 如果系统是只读的, 很容易实现系统分区可容忍性, 因为副本

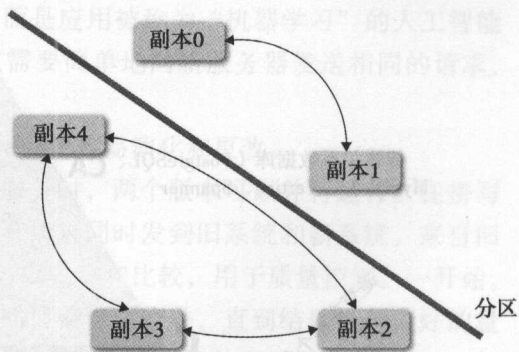


图 1.8 相互分隔的节点

① 真相是, 全球性的 ATM 系统并不要求数据库一致性。这种一致性可能因为网络延迟和故障而遭到挫败。对于银行来说, 与在 ATM 网络不佳时多付出有限数量的现金相比, 让客户因为无法取得现金而感到不愉快的代价更高。欺诈性的交易可以在事后处理。每日取款限额能够避免重大的欺诈。评估平均费用比实施全球一致性数据库容易。

不需要相互通信。但是考虑包含状态的副本示例，更新首先在一个副本上进行，然后复制到其他副本。如果副本无法相互通信，系统就无法保证在固定时间内传播更新，从而成为失效的系统。

现在，考虑两个服务器按照主—从关系协作的情况。两个服务器都维护完整的状态拷贝，如果主服务器失效（通过心跳缺失确定），从服务器接管其任务——也就是说，通过专用网络，进行两台服务器之间的定期健康检查。如果两台服务器之间的心跳网络进行了分区，从服务器不知道原来的主服务器正常工作且无法在心跳网络上通信，就会升级为主服务器。此时有两台主服务器，系统中断，这种情况称作“脑裂”（Split-brain）。

分区存在一些特殊情况。适用 CAP 原则时，数据包丢失被视为临时的系统分区。另一种特殊情况是网络完全中断。即使分区可容忍性最强的系统也无法在这种情况下工作。

CAP 原则声明，只有一种或者两种属性的组合可以实现，而不可能同时实现三个条件。2002 年，Gilbert 和 Lynch 发表了原始推论的形式证明，使其成为一个定理。人们可以认为，为了实现其他两种属性，须牺牲第三种属性。

图 1.9 中的三角形说明了 CAP 原则。传统的关系数据库（如 Oracle、MySQL 和 PostgreSQL）是一致且可用的（CA）。它们使用事务或者其他数据库技术确保更新的原子性，更新要么完全传播，要么完全不进行。这样，它们保证所有用户在同一时刻看到相同的状态。较新的存储系统（如 HBase、Redis 和 Bigtable）专注于一致性和分区可容忍性（CP）。在分区时，它们变为只读或者拒绝响应任何请求，而不是表现出不一致性，允许某些用户看到旧数据，而其他用户看到新数据。最后，Cassandra、Riak 和 Dynamo 等系统专

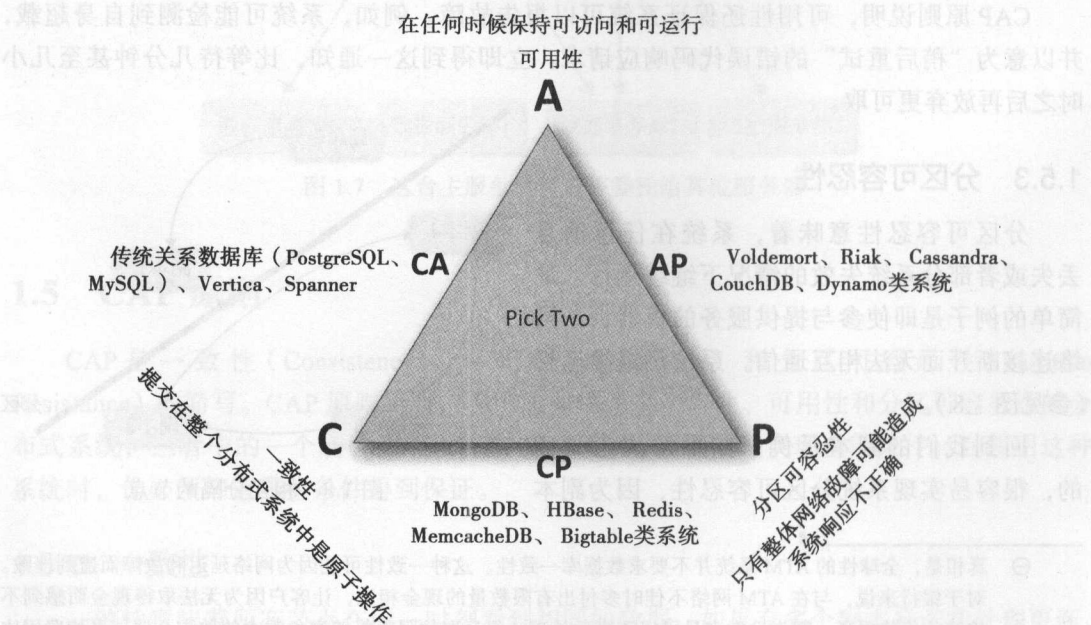


图 1.9 CAP 原则

注于可用性和分区可容忍性 (AP)。它们强调对请求的服务, 即使这意味着某些客户端会接收到过时的结果。这种系统往往适用于全球分布的网络, 每个副本使用互联网等不可靠媒介相互通信。

SQL 和其他关系型数据库使用 ACID 这一术语描述其 CAP 三角形。ACID 是原子性 (Atomicity, 事务“要么全部成功, 要么全部不成功”)、一致性 (Consistency, 每个事务之后, 数据库处于有效状态)、隔离性 (Isolation, 并发的事务得出相同的结果, 就像顺序执行一样) 以及持久性 (Durability, 提交的事务数据在系统崩溃或者发生其他问题时不会丢失)。提供较弱一致性模型的数据库往往自称为 NoSQL, 并描述为 BASE: 基本可用、软状态服务以及最终一致性。

1.6 松散耦合系统

分布式系统应该是高可用、持久且无中断演化及更改的。所有子系统都可以在系统正常运行时替换。

为了实现上述目标, 分布式系统使用抽象 (Abstraction) 来构建松散耦合系统。抽象意味着, 每个组件提供一个接口, 该接口以隐藏实现细节的手段定义。如果每个组件对其他组件的内部细节所知甚少 (或者完全不知), 系统就是松散耦合的。结果是, 一个子系统可以由提供相同抽象接口的另一个系统替代, 即使两者的实现完全不同。

以拼写检查服务为例。好的抽象应该是获取文本, 并返回其中哪些单词拼写错误以及每个错误单词可能更正的列表。不好的抽象则简单地提供一个词典, 供前端查询类似的单词。后一种抽象不好的原因是, 如果发明了拼写检查的全新方法, 使用拼写检查服务的前端都需要重写。如果这个新版本不依赖于词典, 而是应用被称为“机器学习”的人工智能技术, 在“好”的抽象中, 前端不需要更改, 只需要简单地向新服务器发送相同的请求, 而“坏”抽象的用户就没有这么幸运了。

由于这个原因 (以及许多其他的原因), 松散耦合更容易演化和更改。

继续我们的例子, 在准备投放新的拼写检查服务时, 两个版本可以并行运行。在拼写检查系统之前的负载均衡器可以进行编程, 将所有请求同时发到旧系统和新系统。来自旧系统的结果将发送给用户, 而来自新系统的结果可以收集和比较, 用于质量控制。一开始, 新系统产生的结果可能不那么好, 但是随着时间的推移可以改进, 直到结果得到更好的量化评价。那时, 新系统将投产。为了谨慎起见, 可能所有查询中只有 1% 通过新系统——如果没有用户投诉, 新系统将接管更大的部分。最终, 所有响应都来自新系统, 旧系统退役。

其他系统需要的精度和准确度高于拼写检查系统, 例如, 新系统可能要求与旧系统“错误对错误相容”, 然后才能提供新功能。也就是说, 新系统不仅要复制旧系统的功能, 还要复制旧系统的缺陷。在这种情况下, 向两个系统同时发送请求并比较结果对于部署运营任务来说至关重要。

案例研究：改进之前的仿真

Tom 在 Cibernet 工作时，参与了一个替换旧系统的项目。因为这是一个财务系统，新系统必须证明具有“错误对错误相容性”，才能部署。

旧系统构建于 Web 之前的过时技术基础上，已经变得很复杂和僵化，无法添加新功能。新系统构建于更新、更好的技术上，采用了更清晰的设计，更容易适应新功能。这两个系统并行运行并比较结果。

此时，工程师在旧系统中发现了一个缺陷。货币兑换采用了非标准的方式，结果稍有偏差。为了使两个系统的结果有可比性，开发者对这一缺陷进行逆向工程，并在新系统中进行仿真。

现在，旧系统和新系统的匹配结果达到了分的级别。由于公司已经确信新系统的“错误对错误相容性”，新系统被激活，成为主系统，旧系统被禁用。

在这个时候，可以为系统增加新功能并进行改进。不出意外，第一个改进就是删除仿真货币兑换缺陷的代码。

1.7 速度

迄今为止，我们已经详细地介绍了涉及大型分布式系统设计的许多考虑因素。对于 Web 和其他交互式服务，速度可能是最重要的一个项目。获得信息、存储信息、计算和转换信息以及传输信息都需要时间，没有哪一步是立刻发生的。

交互式系统需要快速响应。用户倾向于将快于 200 ms 的速度看成是“即时”的，他们喜欢更快的系统。研究表明，在网站上人为加入 50 ms 的延迟，收入急剧下降。对于批处理和非交互式系统，时间也很重要，这些系统的总吞吐率必须满足或者超过输入工作流。

设计高性能系统的一般策略是用我们对系统处理请求的最佳估算值设计系统，然后构造原型测试我们的假设。如果假设出错，则回到第一步，下一个迭代至少可以得到我们此次实验结果的指导。在我们构造系统时，可以重新计量，如果发现估算和原型不能像希望的那样指导我们，可以调整设计。

在设计过程开始时，我们往往创建许多种设计，估算每种设计的速度，淘汰不够快的设计，我们不能自动选择最快的设计，最快的设计可能比足够快的设计要贵得多。

我们如何确定某种设计是否值得追求？构造一个原型是很费时的，简单的估算可以推导出许多结论。选择几个常见的事务，并将其分解为较小的步骤，并估算每个步骤花费的时间。

最耗费时间的两个因素是磁盘访问和网络延迟。

磁盘访问很慢是因为包含了机械操作。为了从磁盘上读取一块数据，读磁头必须移到正确的磁道，然后，盘片必须旋转直到所需的数据块位于读磁头之下。这一过程通常需要花费 10 ms。相比起来，RAM 读取等量信息的时间只需 0.002 ms，速度比磁盘快出 5000 倍。

磁头和盘片（称为主轴）一次只能处理一个请求，但是，一旦磁头位于正确的磁道之下，可以按顺序读取许多数据块。因此，如果两个数据块相邻，读取两个数据块和读取一个数据块往往几乎一样快。固态硬盘（Solid-State Drive, SSD）没有机械转动盘片，因此速度快得多，但是也更贵。

网络访问之所以慢是因为受限于光速。一个数据包从加利福尼亚州到荷兰大约需要 75 ms，其中一半时间是因为光速。其他延迟可能归因于每个路由器的处理时间、铜线与光纤通信线路之间的转换以及各端打包及拆包的时间等。

在同一网段上的两台计算机似乎可以即时通信，但是实际上并非如此。这种情况下的时标很小，导致其他延迟成为更大的因素。例如，在局域网上传输数据时，第一个字节很快到达，但是接收数据的程序通常等到整个数据包都接收时才处理。

在许多系统上，计算所花费的时间比起网络和磁盘操作的延迟要小得多。因此，如果你知道用户与数据中心的距离以及需要的磁盘寻道次数，就可以估算一个事务所花费的时间。你的估算往往足以抛弃明显不好的设计。

为了说明这一点，想象你在构建一个电子邮件系统，要求能够在 300 ms 内从消息存储系统中读取一条消息并显示。我们将使用图 1.10 中列出的近似时间，帮助设计解决方案。

Google 员工 Jeff Dean 推广如下常见数字图表，以辅助架构和伸缩决策。正如你所看到的，某些选项之间相差许多个数量级。这些数字每年都在缩小，在网上可以找到更新。	
操 作	典型时间
L1 缓存引用	0.5 ns
分支预测错误	5 ns
L2 缓存引用	7 ns
互斥体锁定 / 解锁	100 ns
主存引用	100 ns
用 Zip 算法压缩 1 KB	10 000 ns (0.01 ms)
在 1Gbit/s 网络上发送 2 KB	20 000 ns (0.02 ms)
从内存顺序读取 1 MB	250 000 ns (0.25 ms)
在同一个数据中心内往返传输	500 000 ns (0.5 ms)
从 SSD 读取 1 MB	1 000 000 ns (3 ms)
磁盘寻道	10 000 000 ns (10 ms)
从网络顺序读取 1 MB	10 000 000 ns (10 ms)
从磁盘顺序读取 1 MB	30 000 000 ns (30 ms)
从加州发送数据包到荷兰，再返回加州	150 000 000 ns (150 ms)

图 1.10 每个工程师都应该知道的数字

我们首先从头到尾跟踪这一事务。浏览器的请求可能来自于另一块大陆。该请求必须进行身份验证，查询数据库索引，以确定从哪里获得消息文本，检索消息文本，最后格式化响应并回传给客户。

现在，我们预算自己所不能控制的项目。从加州向欧洲发送一个数据包通常需要 75 ms，在物理学让我们更改光速之前，这是无法改变的。我们的 300 ms 预算需要减去 150 ms，因为不仅要考虑发送请求所花费的时间，还要考虑应答的时间。预算中有一半是我们无法控制的。

我们与身份验证系统运营团队交谈，他们建议为身份验证预算 3 ms。

数据格式化的时间很短，远不及我们估算的其他项目，所以可以忽略。

剩下的 147 ms 用于从存储中检索消息。如果典型的索引查找需要 3 次磁盘寻道（每次 10 ms），读取大约 1 MB 信息（30 ms），那么就需要 60 ms。读取消息本身可能需要 4 次磁盘寻道，读取约 2 MB 信息。这样总共需要 160 ms，超过了我们剩下的 147 ms 预算。

我们是怎么知道这一切的？

我们是如何知道读取索引需要 3 次磁盘寻道的？这需要 UNIX 文件系统内部工作原理的知识：如何寻找目录中文件的索引节点（inode），如何用 inode 寻找数据块。这就是理解操作系统内部原理对设计和运营分布式系统很重要的原因。UNIX 和类 UNIX 操作系统的内部原理都有很好的文档，这是它们相对其他系统的优势之一。

虽然我们对设计不能满足设计参数感到失望，但是我们应该对避免一场灾难感到高兴。现在知道，比一切都为时已晚才发现更好。

对于索引查找，60 ms 似乎是很长的时间，我们可以显著地改善它。如果将索引保存在 RAM 中会怎么样？可能实现吗？简单的计算就能估计出，3 级深度查找树就能扇出到足够的机器，跨越这些数据。树的检索需要 5 个数据包，如果它们在相同的数据中心，大约需要 2.5 ms。新的总耗时（ $150\text{ ms} + 3\text{ ms} + 2.5\text{ ms} + 100\text{ ms} = 255.5\text{ ms}$ ）低于预算的 300 ms。

对于时间敏感的其他请求，我们可以重复这一过程。例如，我们发送邮件的次数少于阅读邮件的次数，所以发送邮件的时间可能不被认为是关键的。相反，删除邮件消息往往和阅读邮件消息的次数一样多。我们可以重复计算几种删除方法，比较它们的效率。

其中一种设计可能联系服务器，从存储系统中删除消息和索引。另一种设计可能只让存储系统在索引中将消息标记为“已删除”。这样做的速度应该明显更快，但是需要一个新元素，负责收集标记为“删除”的邮件消息，并且不定期地压缩索引，删除所有标记为“删除”的项目。

异步设计可以实现更快的响应时间，意味着客户端向服务器发送请求，并且快速地将控制返回给用户，而不等待请求完成。用户感觉这种系统更快，但是实际的工作是滞后的。

异步设计的实现更复杂，服务器必须对请求进行排队，而不是真正地执行操作。另一个进程从队列中读取请求，并在后台执行。客户端也可以简单地发送请求并在以后检查应答，或者分配一个线程或者子进程等待应答。

这些设计都是可行的，但是每种设计的速度和实现复杂度各不相同。有了速度及成本的估算，在原型支持下，就可以做出实现哪一种设计的业务决策。

1.8 小结

分布式计算在许多方面不同于传统计算。它的规模更大；由许多机器各自完成特殊的任务；复制服务以增加容量。硬件故障不作为紧急情况或者异常看待，而是作为系统预期中的一部分。因此，系统必须绕开故障。

大型系统通过组合较小的部件构建。我们讨论了3种典型的构成：有许多后端副本的负载均衡器、具有许多不同后端的前端以及服务器树。负载均衡器在许多复制的系统间划分流量。具有许多不同后端的前端并行使用不同的后端，每个后端执行不同的进程。服务器树使用树形配置，树的每个级别服务于不同的目的。

不管是不断更新信息的大型数据库还是许多系统需要不断访问的少数关键状态位，分布式系统的状态维护都很复杂。CAP原则说明，构建同时保证一致性、可用性和分区容忍性的分布式系统是不可能的。3种特性中最多只能实现两种。

系统应该是随着时间推移而演化的，为了使这种演化更容易发生，组件应该是松散耦合的。每个组件体现了所提供服务的抽象，这样，可以在不改变抽象的情况下替换或者改进内部逻辑，因此，除了从新功能中获益之外，对服务的依赖性不需要改变。

分布式系统的设计需要理解各种操作运行所花费的时间，才能设计符合延迟预算的时间敏感处理。

练习

1. 什么是分布式计算？
2. 描述分布式计算的3种主要构成模式。
3. 存储状态的3种模式是什么？
4. 有时候，主服务器对某个问题没有做出应答，而是以答案所在位置作为应答。这种方法有什么好处？
5. 1.4节描述了分布式文件系统，包含一个读取TB级数据的示例。简述TB级数据的写入工作原理。
6. 解释CAP原则。（如果你认为CAP原则很了不起，可以阅读《The Part-Time Parliament》(Lamport & Marzullo, 1998) 和《Paxos Made Simple》(Lamport, 2001)。

7. 系统松散耦合意味着什么？这种系统的好处是什么？
8. 给出你所经历过的松散和紧密耦合系统示例。是什么使得它们松散或者紧密耦合？
9. 如何估算系统处理请求（如检索电子邮件消息）的速度？
10. 在 1.7 节中，介绍了 3 种处理电子邮件删除请求的设计思路。估算 3 种设计中删除一个邮件消息请求所花费的时间。首先概述每种设计所需的请求处理步骤，然后将其分解为单独的操作，直到可以做出估算。

小结 8.1

我们与身份验证系统运营团队交谈，他们建议为每个删除请求分配 3 ms。我们假设删除请求由客户端发出，这更可能是由客户端而不是由服务器发起。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

对于时间敏感的其他请求，我们可以重复这一过程。例如，我们假设邮件的下载次数为 10，那么就需要 10 个删除请求。删除请求的响应时间为 160 ms，那么就需要 1600 ms 来删除邮件。对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

讨论

对于时间敏感的其他请求，我们可以重复这一过程。例如，我们假设邮件的下载次数为 10，那么就需要 10 个删除请求。删除请求的响应时间为 160 ms，那么就需要 1600 ms 来删除邮件。对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

对于删除请求，我们假设邮件存储在数据库中，而不是存储在服务器上。删除请求的响应时间包括网络延迟、服务器处理时间、数据库查询时间、数据库更新时间和网络延迟。假设网络延迟为 10 ms，读取大约 1 MB 信息（30 ms），那么就需要 40 ms 来读取邮件。数据库更新时间为 10 ms，那么就需要 50 ms 来更新数据库。网络延迟为 10 ms，那么就需要 60 ms 来发送响应。总时间为 160 ms。

为运营而设计

你的工作是设计能够运作的系统。

——Theo Schlossnagle

本章对最常用的运营任务进行分类，讨论如何为它们进行设计，还要讨论如何改造没有考虑运营的现有架构设计。

为运营而设计意味着确保所有常规运营职能都能很好地实施。常规运营职能包括定期维护、更新和监控等任务。这些问题必须在规划的早期阶段就了然于心。

在考虑指定服务的整个生命期时，生命期中只有一小部分用于构建服务的特性。生命期的绝大部分时间花费在服务的运营上，但是传统上即便考虑了软件的运营职能，其优先级也被认为低于功能需求。

提供高可用性服务的最佳策略是在软件中植入改善人们执行和自动化运营任务能力的特性。这与将运营作为后期考虑，运营工程师被迫处于“运行其他人构建的系统”的情况形成了对比。后一种想法已经过时了。

2.1 运营需求

软件通常是根据与最终用户所见和所做相关的需求设计的，很少考虑顺利运营所需的功能。因此，系统管理员发现自己对关键交互缺乏控制点。当我们为运营而设计时，会考虑基础设施生命期的常规职能，包括（但不限于）如下方面：

□ 配置

□ 启动和关机

- ☐ 队列排空
- ☐ 软件升级
- ☐ 备份与恢复
- ☐ 冗余性
- ☐ 数据库副本
- ☐ 热切换
- ☐ 单独功能开关
- ☐ 优雅降级
- ☐ 访问控制和速率限制
- ☐ 数据导入控制
- ☐ 监控
- ☐ 审计
- ☐ 调试设施
- ☐ 异常收集

配置和备份 / 恢复等特性使典型的运营任务成为可能，队列排空和单独功能开关等特性使这些任务能够无缝完成。正如前面在 1.1 节中所讨论的，许多这类功能形成了调试、调整和维护大型系统所需要的内省特性。

一般来说，开发人员和经理们很少思考这些问题，他们往往排除需求列表，至多在事后考虑或者假定运营团队会依靠即兴发挥“想出某种办法”。事实是，这些任务很重要，没有特定的功能就无法很好地完成（甚至完全无法完成）。在最坏的情况下，系统的设计不利于“即兴创作”解决方案。

有些组织使用“非功能性需求”，而不使用“运营需求”这一术语。我们认为那是一个误导性的术语。虽然这些特性没有直接负责应用程序或者服务的功能，但是“非功能性”这一术语的隐含意思是这些特性不包含某种功能。没有这些特性的支持，服务就无法存在；它们是必不可少的。

本章剩下的部分讨论这些运营特征以及实现它们的特性。对于具备运营经验的人来说，许多特征似乎都是显而易见的。但是，在此列出这些特征，是因为我们曾经观察到至少一个系统因为忽略了它们而受到惩罚。

2.1.1 配置

系统必须可用自动化手段配置，包括初始配置和之后的更改，必须能够执行如下任务：

- ☐ 建立配置的备份并恢复
- ☐ 查看存档拷贝和另一个修订版本之间的差别
- ☐ 在不停机的情况下存档运行配置

实现上述所有目标的典型途径是配置采用具备严格定义格式的文本文件形式。自动化

系统很容易生成这种文件。文本文件容易解析，因而可以审计，也很容易存档。它们还可以保存在源代码存储库中，用标准文本比较工具（如 UNIX 的 diff）分析。

在某些系统中，配置在系统运行时动态更新。这种“状态”可以反映到主配置文件，或者单独的实体上。在这种情况下有额外的需求。

必须有自动化读取和更新状态的手段。这一步骤可能通过 API 或者配置文件的读取/更新完成。如果使用文件，必须存在一个加锁协议，避免服务和外部自动化手段以不完整的状态读取文件，并避免更新冲突。必须有在不同于活动系统上激活配置的情况下进行配置健康检查的工具。

使用难懂的二进制大对象（BLOB）作为配置文件是较差的选择，这种文件是人类无法阅读的。在这类系统中，不可能保存配置的历史、随时查看更改。在更改很小而无法记忆，但是又足以造成某种问题时，这些奇怪的问题往往要通过分析配置文件的更改来调试，如果文件不是普通文本，这类分析不可能进行。

我们曾经被提供提取完整配置的 API，结果却不是真正完整配置的系统搞得焦头烂额。很多时候，只有在灾难恢复演练或者紧急情况下才发现这种遗漏。因此，每个新版本应该经过测试，验证配置数据中没有任何遗漏。

从运营的角度，理想状况下，配置由一个或者多个容易检查、存档和比较的普通文本文件组成。

有些系统直接从源代码存储库读取其配置，这很方便，也是强烈建议的方式。然而，必须可以禁用这一功能并直接提供配置。这种方法可以在紧急情况下（源代码库故障）和测试时使用。禁用存储库配置功能的使用必须以某种方式曝光，使监控系统能够检测到。然后，其他用户可以意识到它的发生，例如，在仪表盘上显示这种状态。这也可能是一个可触发警报的事件，在那种情况下，如果这一功能被用于生产系统，可以生成警报。在功能禁用超过某一时间的时候发出警报，可以确保临时修复不被遗忘。

简单的配置不需要 GUI

来自 IBM 的一位产品经理曾经告诉 Tom，该公司已经花费重金在系统管理工具中加入了图形用户界面（Graphical User Interface, GUI），以简化配置。令该团队沮丧的是，大部分客户不使用 GUI，因为他们已经编写了生成配置文件的 Perl 脚本。

2.1.2 启动和关机

机器开机时，服务应该自动启动。如果机器正常关机，系统应该包含相应的操作系统（OS）钩子，正常关闭服务。如果机器突然崩溃，下一次启动系统将在提供服务之前自动执行数据校验或者修复。

确保服务在机器重启之后启动很简单，只需要安装一个启动时脚本，或者使用监控并重启进程的系统（如 Ubuntu Upstart）。也可以使用完整的进程管理系统，如 Apache Mesos

(Metz, 2013) 或 Google Omega (Schwarzkopf, Konwinski, Abd-El-Malek & Wilkes, 2013), 这些系统不仅在机器重启时启动进程, 还可以在机器死机时在完全不同的机器上重启该进程。

启动或者关闭系统所需要的时间应该记入文档。这对于灾难恢复准备来说是必需的。人们必须知道系统安全关闭的快慢, 才能计划不间断电源 (Uninterruptible Power Supply, UPS) 系统的电池容量。大部分 UPS 电池能够保持供电大约 5 分钟。在停电之后, 启动数千台服务器可能非常复杂。知道预计的启动时间和规程, 可以显著地缩短恢复时间。

测试所有系统同时断电时系统的表现很重要。这是数据中心常见的压力来源。数千个硬盘电机同时旋转产生巨大的电源消耗, 可能使供电系统超载。一般来说, 在第一次尝试时可以预期 1%~5% 的机器无法启动。在包含 1000 台机器的系统中, 恢复全部机器可能需要一个大的团队。

与此相关的概念是“仅崩溃”(Crash-only) 软件。Candea 和 Fox (2003) 观察到, 大部分系统中, 系统崩溃后的恢复规程对于系统可靠性至关重要, 但是接受的质量保证 (Quality Assurance, QA) 测试却太少。具备高可用性的服务应该很少使用顺序关机过程。为了保持恢复规程重要性与接受测试次数的平衡, 这些作者提出, 不要采用顺序关机或者顺序启动规程。因此, 停止软件的唯一方式是使其崩溃, 启动软件的唯一方式是演练崩溃恢复系统。这样, 崩溃恢复过程频繁演练, 测试过程也不太可能忽略它。

2.1.3 队列排空

必定有一个顺序关闭过程, 在它触发时停止系统服务进行维护。排空 (Drain) 发生在某个服务被告知停止接收新请求, 只完成所有“在途”请求时。这有时候被称作“跛脚鸭”模式 (Lame-Duck Mode)。

这种机制在使用 1.3.1 节描述的“具有多个后端副本的负载均衡器”时特别重要。实施软件更新时, 每次卸下一个副本, 升级, 然后恢复服务。如果简单地“杀死”每个副本, 任何在途的请求将被丢失。采用“排空”模式更好, 这样副本可以继续处理请求, 但是有意地使负载均衡器的健康检查请求失败。负载均衡器发现健康检查失败时, 将停止向该副本发送新请求。在一段时间内没有接收到新请求且现有请求完成后, 就可以安全地“杀死”副本, 执行升级。

清空队列

在开发开创性的 Palm VII 无线消息服务时, 该团队意识到主应用程序没有排空能力。关闭它的任何尝试都会丢失处理中的所有消息。Strata 协商加入这项功能, “排空并退出”功能使运营团队可以在不丢失消息的情况下, 关闭服务器进行维护, 或者置换它们。

类似地, 可以在排空模式中启动服务很有用。在这种情况下, 负载均衡器将不向该副本发送新的流量, 但是运营团队可以直接向它发送消息以供测试。一旦确认, 副本退出排

空模式，向负载均衡器发出发送流量的信号。

2.1.4 软件升级

软件必须可以在不停止服务的情况下升级。软件通常在负载均衡器之后，通过置换副本进行升级。

有些系统可以在运行时升级，这风险更大，需要精心的设计和大量的测试。

非复制型系统难以在不停机的情况下升级。唯一的替代方法往往是克隆该系统、升级克隆副本，在客户尚未注意到的时候将新升级的系统置换到位。这通常是一种高风险——有时候是临时的——解决方案。生产系统的克隆可能是不完善的拷贝，因为难以确定克隆在没有任何更改的时候精确地进行。

2.1.5 备份和恢复

必须可以在系统运行的时候备份和恢复服务的数据。

旧的系统往往需要停止服务才能备份或者恢复。这种方法对于只有 10 个用户的小办公室可能足够，但是对于具有数千或者数百万用户的网站来说并不合理。允许实时备份和恢复的系统设计则大不相同。

实现不干扰服务实时备份的方法之一是在数据库的一个只读副本上备份。如果系统可以动态添加和删除副本，则将副本从服务中删除、冻结并用于建立备份。然后，这个副本被添加回系统。使用专用于这一过程的特殊副本是常见的做法。

实时恢复往往通过提供特殊 API、在恢复操作过程中插入数据来完成。这种架构应该允许单一账户的恢复，最好是不要将用户或者用户组挡在服务之外。

例如，电子邮件系统应该在不恢复所有账户的情况下，恢复单个用户的账户。它应该实时进行这项操作，使用户在恢复消息出现时可以继续使用服务。

备份和恢复都给系统带来额外的负载。在容量规划（动态余量）和延迟计算中必须考虑这一负担。

2.1.6 冗余性

许多可靠性和伸缩技术的基础是运行服务的多个冗余副本的能力。因此，服务的设计应该支持这些配置。服务副本已在 1.3.1 节中讨论，在副本之间复制状态的难题已在 1.5 节中讨论。

如果服务没有设计为在负载均衡器之后工作，它的有效性仅仅靠“运气”，这不是系统管理的合理方式。只有最基本的服务可以在这种情形下工作。这种系统很可能无法正常工作，甚至更糟——看似正常工作，但是之后却发展出难以跟踪的问题。

常见的一个问题是，Web 服务器在本地保存用户的登录状态，但是没有传达给副本。当负载均衡器未来从同一个用户接收请求时，如果这些请求发送给不同副本，该用户将会

被要求再次登录。除非用户已经在每个副本中登录过，否则这种情况会重复出现。该问题的解决方案在 4.2.3 节中讨论。

2.1.7 数据库副本

访问数据库的系统必须支持数据库的伸缩性。在分布式系统中，实现数据库访问伸缩性的最常用方法是创建一个或者多个只读副本。主服务器实施所有更改数据库的事务，然后，更新通过批量传输传递给只读副本。服务可以正常访问主数据库，如果查询不做任何更改，则发给只读副本。大部分数据库访问是只读的，所以大部分工作负担分摊到副本上。这些副本提供快速（但是稍微过时）的访问。主数据库提供对“最新鲜”数据的完整服务，但是可能较慢。

使用数据库的软件必须经过特殊的设计，才能支持只读副本。它不打开一个数据库连接，而是创建两个连接：一个连接到主数据库，另一个连接到其中一个只读副本。开发人员编码每个查询时，认真考虑查询发送到哪一个连接将在速度和数据新鲜度上求得平衡，清醒地意识到每个直接发送到主数据库的查询都将消耗其宝贵的资源。

即使在数据库没有任何副本、两个连接都指向相同服务器时，分离这些查询也是一种好的做法。某一天，你可能想要添加只读副本。确定查询使用的连接最好在一开始构建查询时完成，而不是在几个月或者几年以后。话虽如此，如果在事后对系统进行翻新，最好花点时间识别可以转移到只读副本的重要操作，而不是检查源代码中的单个查询。另外，可以为某些特定的用途（如备份）创建一个只读副本。

倒霉的只读副本

在 Google，Tom 体验过数据库主机与其副本之间的竞争情况。经过认真的设计，系统向数据库主机发送写入指令，所有读操作均在副本进行。但是，某个组件在数据更新之后立刻读取，往往因为看到来自副本的过时信息而困惑。由于该团队没有时间重新编码这个组件，只好重新配置组件，将写入连接和读取连接都指向主数据库。尽管在许多读查询中只有一个必须发送到主数据库，但是所有查询实际上都发往主数据库。由于这个组件一天只使用一两次，所以没有给主数据库造成无法承受的负担。

随着时间的推移，使用模式出现了变化，这个组件的使用更加频繁，最终整天都在使用。一天，由于这个组件的负载使主数据库超载，出现了停机故障。在那个时候，这个组件不得不重新设计，以便正确地分离查询。

对于依赖运气而不是开发人员官方支持的人们，还有一个警告：运气总会用完的。今天依靠运气，可能在下一个软件发行版本上造成灾难。即使对于运营人员来说可能是一个长期的策略，但是支持这种配置对于开发人员也将是一个“意外的请求”。开发人员可以正当地拒绝修复该问题，因为这从一开始就不是受到支持的配置。你可以想象这种混乱的情况和造成的冲突。

2.1.8 热切换

服务组件应该可以换入或者换出其服务任务，而不会造成整体服务故障。软件组件可以自行完成热切换，也可以兼容于某种负载平衡器或控制这一过程的其他重定向服务。

有些物理组件（如电源或者磁盘）可以在加电（或“热”）的情况下更换。**可热切换设备**可以在不影响机器其余部分的情况下更换。例如，电源可以在不停止操作的情况下替换。**可热插拔设备**可以在机器运行时安装或者拆卸。在执行这种操作之前或之后可能需要管理任务。例如，除非告诉操作系统扫描新磁盘，否则硬盘可能不会被识别。新网卡可能会被发现，但是除非专门编程为定期扫描新接口，否则应用程序服务器软件在没有重启之前可能不会看到它们。

供应商所称的“可热插拔”和“可热切换”的含义往往不清晰。直接测试系统以理解这一处理的结果，以及应用程序级软件的响应。

2.1.9 单独功能开关

应该提供一个配置设置（开关，toggle）以启用或者禁用每个新功能。这可以使新软件发行版本的试运行独立于用户对新功能可用的时间。例如，如果一个新功能将在周三中午出现在网站上，在准确的时间“推送”一个新的二进制文件会难以协调。但是，如果每个功能可以独立启用，该软件可以提前部署，在预期的时间更改一个配置设置启用新功能。这常被称为**标志切换**（flag flip）。这种方法对于处理造成问题的新功能也很有用，通过标志切换禁用单独功能，比回滚到前一个二进制代码更容易。

更高级的开关可以为特定用户组启用，某项功能可以为一小组信任的测试者启用，这些测试者可以提早使用该功能。一旦经过检验，可以利用这个开关为所有用户启用功能，也可以为后续的更大群体启用该功能。详见 11.7 节。

2.1.10 优雅降级

优雅降级（Graceful degradation）意味着，软件在超载或者所依赖的系统停机时有不同的表现。例如，网站可能有两个用户界面：一个比较丰富且充满图形，而另一个则是轻量级的纯文本界面。用户在正常情况下接收到丰富的界面，但是，如果系统超载或者有触及带宽限制的风险时，它会切换为轻量级模式。

优雅降级还要求软件在出现故障时有智能的表现。如果数据库停止接收写入（检测到数据破坏时常见的管理性防范措施），服务可以变为只读。如果数据库完全无法访问，软件利用缓存工作，使用户能够看到部分结果，而不是错误信息。

当服务不再能够访问所依赖的服务时，相关功能可以消失，而不是显示一个损坏的网页，或者“404 页面未找到”错误。

即使小站点也知道，提供一个临时 Web 服务器，对任何查询都显示“网站正在建设中”，比让用户完全接收不到任何服务更好。对于这种情况有一些简单的 Web 服务器软件

包可用。

案例研究：Google 应用的优雅降级

Google Docs（谷歌文档）部署了许多优雅降级技术。在只有只读数据库副本可用时，Google 的字处理器可以切换到只读模式。如果服务器无法访问，客户端 JavaScript 可以使用浏览器中的缓存数据。Gmail 提供丰富的基于 JavaScript 的用户界面，以及较为简洁的仅 HTML 界面，在必要时自动显示。如果整个系统都无法访问，用户会看到一个通用的首页，显示系统状态，而不是毫无反应。

2.1.11 访问控制和速率限制

如果服务提供一个 API，API 应该包含访问控制列表（Access Control List，ACL）机制，确定哪些用户允许或者拒绝访问，还要确定速率限制设置。

ACL 是一个用户列表，以及表示他们是否得到访问系统授权的信息。例如，可以对某些互联网协议（IP）地址或者地址块、某些用户或者进程，或者通过其他标识机制限制访问。IP 地址是最弱的标识方式，因为它们很容易伪造。应该使用更好的标识机制，如使用数字证书证明身份的公钥基础设施（Public Key Infrastructure，PKI）。

最简单的 ACL 是允许访问的用户列表，其他所有用户都被禁止。这称之为默认关闭（default closed）策略，该列表称作白名单。相反的机制是默认开放（default open）策略，除了出现在黑名单上的用户，其他所有用户默认可以访问。

更高级的一种 ACL 是注明“允许”或者“拒绝”的用户或用户组有序列表。如果 ACL 中没有提及某个用户，默认的操作可能是允许用户（失效开放，fail open）或者拒绝用户（失效关闭，fail close）。

除了表示许可之外，ACL 还可以指明速率限制。不同用户可能允许不同的每秒查询次数（QPS），超出速率限制的请求将被拒绝。例如，服务可以为高级客户提供无限制的 QPS 速率，为定期付费客户提供中等 QPS 速率，而对于未付费客户提供低 QPS 速率甚至完全不能访问。

案例研究：Google 的 ACL

Google 所有内部 API 使用的远程过程调用（Remote Procedure Call，RPC）协议有强大的 ACL 系统。连接通过 PKI 验证，以便确保知道服务客户端的身份以及所属的组。身份和用户组是全局定义，代表与单独外部客户相对的产品和服务。ACL 指定单独用户或者用户组的权限：允许、拒绝或者包含速率限制的允许。各个团队协商某个服务的访问 QPS 速率，作为该服务容量规划的一部分。没有协商速率的团队也能够访问，但是速率限制很低。这使得所有团队都可以尝试新服务，消除了服务团队协商数百个轻量级或者偶发请求所需的工作负担。

2.1.12 数据导入控制

如果一个服务定期导入数据,应该确立允许运营人员控制数据接收、拒绝或者替换的机制。输入数据的质量各不相同,导入数据的系统需要一种限制实际导入数据,以便忽略已知低质量数据的手段。如果某个坏的记录导致系统问题,人们可以通过配置拦截它,而不用等待软件更新。

这种系统使用前面看到的白名单/黑名单术语。黑名单是规定拒绝输入的手段,假设所有其他数据都可以接收。白名单用于规定接收的数据,其他所有数据都被拒绝。

除了控制输入数据流之外,我们需要用本地提供的数据补充输入数据源的手段。这使用导入数据扩增文件的方式实现。

确定更改限制也能够避免问题的发生。例如,如果每周导入数据更改通常少于所有记录的20%,如果更改会影响30%或者以上的记录,操作者需要人工批准。这可以避免软件缺陷或者质量不良的新数据批次造成的灾难。

案例研究: Google Maps 本地商户列表

Google 从各种“本地商户目录”服务中订阅商户列表,用于其地图相关产品。这些信息提供商定期发送必须处理和导入地图系统的数据。这些数据的质量令人失望:列表经常不正确、损坏或者毫无用处。向提供商报告的更改可能需要花费几个月才能出现在数据中。有些来源提供的数据在某些州和国家中的质量不错,但是在其他地区则质量低下。因此,导入这些数据的系统有一个白名单、一个黑名单和一个补充文件。

白名单指出所包含的地区,用于特定提供商的信息可能仅在某些地区具有高质量的情况。一旦验证了特定地区的数据质量,则将其加入白名单。如果质量下降,则删除该地区的数据。

黑名单标识已知的不佳记录,包含了在过去的批次中被标识为“坏”或者“不正确”的记录。

来自商户目录的数据由 Google 独立制作的数据补充。这包括 Google 自己收集的信息,Google 独立更正以覆盖已知的错误数据,以及“复活节彩蛋”(包含在列表中的玩笑)。

2.1.13 监控

运营部门需要对系统工作情况的可见性。因此,系统的每个组件必须向监控系统输出指标。这些指标用于监控可用性和性能,用于容量规划,也是故障检修的一部分。

第16章和第17章将详细介绍这一主题。

2.1.14 审计

日志记录、权限和角色账户的设置使服务可以接受检查,通过安全和依从性审计。这

一领域的变化很快，所以最好始终咨询法律部门，获得最新的法律信息。企业最关心的是如何围绕监管商业的相关法律依从性使用公共云服务：如果选择使用公共云服务，他们会不会无法通过下一次审计，从而面对巨额罚款，或者在通过审计之前无法开展业务？

虽然本地劳工法规通常不会直接影响依从性或者治理问题，但有些国家坚信来自其他国家的人员从事 IT 管理可能是一个依从性问题。考虑如下的情况：系统管理员坐在新泽西州的奥朗日，对德国法兰克福的服务器进行管理。这位系统管理员对当地劳工法规或者欧盟（EU）数据保护条令一无所知，按照计划和批准的更改在整个公司内移动一个虚拟服务器。这种情况下，系统管理员可能违反了至少两项 EU 监管要求，在无意之中使她的雇主违反了依从性要求，遭到制裁或者罚款（甚至两者兼有）。

具有特殊 IT 审计要求的监管法规的例子包括 SOX、j-SOX、c-SOX、PCI DSS、欧盟数据保护条令和新加坡的 MAS。在全世界可以找到 150 多个这类的监管要求。此外，有些全球标准适用于不同的治理环境，如 CobiT 5 和 ISO/IEC 27001、27002 和 27005。IT 指令和依从性在本系列丛书的第一卷中有更完整的介绍（Limoncelli、Hogan & Chalup, 2015）。

2.1.15 调试设施

软件必须生成调试中使用的日志。这些日志应该既是人类可读的，又是可以由机器解析的。此类日志与审计所需的日志不同。调试日志通常记录发送到重要函数调用以及从函数中返回的参数。“重要”的含义也各不相同。

大型系统应该允许在单独模块上启用调试日志。否则，信息量可能难以承受。在某些软件方法中，任何日志信息都必须与指定消息含义和使用方法的文档相匹配。消息和文档必须翻译为系统支持的所有（人类）语言，必须得到市场、产品管理和法律人员的批准。这种策略很快就会导致官僚主义，从而影响生产率。调试日志豁免这些规则，因为这些消息不可见于外部用户。每个开发人员应该主动加入他们认为合适的任何调试日志信息。如何消费这些信息的文档就是源代码本身，运营人员应该能够得到它们。

2.1.16 异常收集

当软件产生异常时，应该集中收集以供分析。软件异常是导致程序退出的严重错误。例如，软件作者可能确定某种特定情况不太可能发生，且难以从这种情况中恢复。因此，程序在这种情况下宣布异常并退出。有些数据破坏的情境由人处理比由软件处理更好。如果你曾经见过操作系统“恐慌”或者发生“蓝屏死机”，那就是异常。

在设计具有可操作性的软件时，往往会使用某种检测异常、收集错误消息和其他信息并提交给集中化数据库的软件框架。这种框架称为异常收集器。

异常收集系统提供 3 种好处。首先，因为大部分软件系统都有某种自动重启能力，可能忽视某些异常。如果你从未发现这些异常的发生，当然就无法处理其潜在根源。但是，异常收集器可以使不可见变成可见。

其次，异常收集帮助确定系统的健康状况。如果有许多异常，应该撤销新软件发行版本试运行等维护工作。如果异常在试运行中急剧增加，可能表明该发行版本质量不佳，应该停止试运行。

异常收集器的第三个好处是可以研究异常历史，找出趋势。研究中发现的简单趋势之一是异常数量的急剧上升或者下降。异常水平通常与特定的软件发行版本相关。另一个需要寻找的趋势是重复。如果记录到特定类型的异常，就能知道它的发生频率上升或者下降情况。如果发生的频率降低，意味着软件的质量得到改善。如果频率上升，就有机会发现并在成为更大问题之前修复根源。

2.1.17 运营文档

开发人员和运营人员应该一起工作，建立服务运营规程的“剧本”。剧本添加了来自更大业务视野中的运营步骤，补充了开发人员编写的文档。例如，开发人员可能编写系统故障切换到热备系统的精确步骤。剧本可以说明，进行故障切换时应该通知谁，故障切换前后必须进行哪些附加检查等。关键是每个规程都必须包含验证成功或者失败的测试套件。下面是一个数据库故障切换规程的示例：

- 1) 通知数据库团队和经理团队的邮件列表，即将发生故障切换。
- 2) 验证热备系统有至少 10TB 空闲磁盘空间。
- 3) 验证这些相关系统都工作于如下参数：(服务器控制面板链接) (数据供应控制面板链接)
- 4) 用系统故障切换规程(链接)执行故障切换。
- 5) 验证相关系统已经成功切换到热备，且运营正常。
- 6) 回复刚才发送的所有有关操作成败的电子邮件。

将服务的基本运营规程记入文档不是开发或者运营团队的单方面职责，而必须是一项协作性的工作，运营团队确保他们所能预见到的运营情境都得到处理，开发人员确保文档覆盖所有代码中可能出现的错误情况，以及使用支持工具和规程的方法及时机。

文档是自动化的跳板。新的过程可能会频繁更改，当它们稳定时，可以识别出好的自动化候选方案(参见第12章)。编写文档还有助于理解哪些方面容易自动化，哪些方面难以自动化，因为记录文档意味着详细地解释各个步骤。

2.2 为运营实现设计

为运营设计的功能必须由某人实现，它们不会魔术般地出现在软件中。对于任何项目，都会发现软件可能全部、部分或者完全没有实现本章列出的功能。在软件中得到这些功能有4种主要途径：

- 从一开始就构建这些功能。

- ❑ 在确定功能时发出请求。
- ❑ 自行编写这些功能。
- ❑ 与第三方供应商合作。

2.2.1 从一开始就构建功能

在这种情况下，敏锐的开发和运营团队在用于运行服务的产品中构建了这些功能。除了有经验的大企业（如 Google、Facebook 或者 Yahoo）之外，极少遇到这种情况。

如果你运气好，能够参与系统的早期开发，那么就应该与开发人员合作，帮助他们排定优先级，使这些功能从一开始就“融入”。如果推动需求的业务团队能够认识到运营需求，那么也能够提供帮助。

2.2.2 在确定功能时提出请求

服务很有可能不包含所需的全部运营功能，因为不可能在系统运营之前知道所有需要的功能。如果你能够接触到开发人员，可以随时请求这些功能。首先，毫无保留地提出你的运营需求，对每个遗漏的功能提交功能申请。功能申请应该标识需要解决的问题，而非具体的实现。列出对业务的风险和影响，使你的请求得到优先安排。在开发人员实现功能时与他们一起工作，可以接受开发人员的咨询，并且鼓励他们。

开发人员的时间和资源都有限，所以排定申请的优先级很重要。优先级排定的策略之一是选择所需工作量最小、影响最大的项目。图 2.1 中的 x 轴是更改的预期影响（从低到高）。y 轴代表更改所需的工作量，也是从简单（低工作量）到困难（高工作量）。专注于最简单的工作（伸手可得的果实）是很有诱惑力的，但是这往往将资源浪费在影响很小的简单任务上。结果可能在情绪上得到满足，但是不能解决运营问题。相反，你应该专注于高影响力的项目，从工作量较低的项目开始逐步选择需要较大工作量的项目。

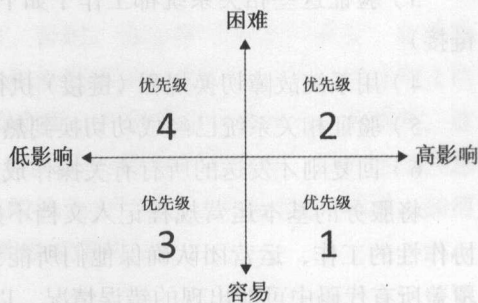


图 2.1 为运营实现设计的优先级

修复最大的瓶颈，通常作用最大。这一要点将在 12.4.3 节中更详细地讨论。

在高效能团队和低效能团队之间发现的差别之一就是高效能团队专注于影响力。

案例研究：运营功能的 80/20 法则

Tom 在 Lumeta 工作时，对于开发人员在运营问题上和新功能上应该花费多少时间产生了分歧。产品经理提出了一个非常有创意的解决方案。产品交替推出大发行版本和小发行版本。大发行版本应该包含重大的新功能。小发行版本应该修复前一重大发行版本的缺陷。

经过协商，在大发行版本中，开发人员在运营团队请求的问题上花费 20% 的时间。小发行版本的意图不是增加重要的新功能，但是包含一个或者两个高优先级的功能。因此，对于小发行版本，开发人员时间的 80% 花费在运营请求上。由于这些发行版本较小，运营请求在大发行版本和小发行版本上花费相同的时间。

2.2.3 自行编写功能

当开发人员不愿意添加运营功能时，自行编写这些功能是一个选择，这是一个不好的选择，原因有二。

首先，开发人员可能不接受你的代码。作为局外人，你不知道他们的编码标准、内部基础架构以及未来软件架构的总体愿景。代码中的任何缺陷都将受到夸张的责备。

其次，这会造成不好的先例，传达“开发人员不需要关心运营功能，因为只要延迟足够长的时间，你就会自己编写它们”的信息。

运营人员应该花费时间编写创建服务运行生态系统的运营服务。编写让开发人员可以用于改善运营的框架。例如，编写一个可以链接的库，简化向监控系统报告状态的过程。编写让开发人员自力更生的工具，而不是让他们依赖于运营团队。例如，编写一个工具收集异常和核心转储以供分析，而不是在需要这个步骤时发电子邮件给运营团队。

这条规则也有例外。局外人提交的代码如果规模较小，更容易接受。在高协作性的组织中，人们可能习惯于接受许多来源贡献的代码。这在开源项目中很典型。在 Google 有一个代码批准过程，使局外人更容易为项目做贡献，并确保代码符合团队标准。这一系统允许反馈和修订，直到代码更改得到双方的接受。只有这样，代码才能进入系统。这一系统还有助于企业内部代码质量和风格保持高标准。在这种系统中，代码质量和编码风格可能与其他团队兼容。

在可能的时候，运营人员应该和开发人员在一起，以便研究代码库，熟悉发行和测试过程，并建立与代码的关系。但是，即便如此也不能代替开发人员添加的运营功能。因此，更好的做法是开发人员和运营人员在一起进行 6 个月的轮换，以便理解运营部门对这些功能的需求。

最有效的方案可能取决于组织的大小和所运营系统的规模。例如，高度的协作在小组织中更容易实现。

2.2.4 与第三方供应商合作

与第三方供应商合作类似于与自己的开发团队合作，需要遵循许多相同的过程，例如提交缺陷、召开定期会议讨论功能申请。

始终以建设性的方式升级问题的可见性，因为供应商对产品受到的批评很敏感。例如，撰写包含功能申请的事后剖析报告，使供应商能够了解申请的背景（编写事后剖析报告的更

详细介绍参见 14.3.2 节)。

如果供应商对申请反应迟钝,你可以编写代码,围绕供应商的软件构建框架。例如,可以围绕不能很好地完成这些任务的供应商软件创建一个包装器,以清晰的方式提供服务的启动和关闭功能。我们强烈建议向外部发布这些系统,作为开源产品。如果你需要它们,其他人也需要。开发一个以你的代码为中心的社区,可以降低支持工作对自身努力的依赖。

2.3 改善模型

好的运营设计可以简化运营。出色的运营设计有助于完全消除某些运营任务,这是一种力量倍增器,往往等同于雇佣了额外的人员。只要有可能,坚持创建在过程中嵌入知识或者能力的系统,代替运营干预的需求。之后,运营团队的工作就从执行重复的运营任务转为构建、维护和改善处理这些任务的自动化措施。

Tom 曾在一个资源分配通过电子邮件申请并人工处理的环境中工作。某个 API 可用之后,整个过程就变成自助式的,用户可以管理自己的资源。一些配给系统能够指定每项“工作”所需的 RAM、磁盘和 CPU。更好的系统完全不需要指定任何资源:它监控使用情况并分配所需资源,在一个群集上的所有工作中维持有效的平衡,并且随时重新分配和移动资源。

常见的运营任务之一是未来容量规划。也就是预测 3~12 个月内需要多少资源。这是一项繁重的工作,作为替代,经过深思熟虑构建的数据收集及分析系统可以进行这些预测,关于容量规划的更多信息参见第 18 章。

创建警告阈值并进行微调是一件永不停息的工作,如果监控系统设置自己的阈值,这项工作就可以消除。例如,一个网站为当年的每个小时应该接受的 QPS 数量开发一个精确的预测模型。然后,系统管理员可以设置在实际 QPS 超过或者低于预测值 10% 以上时发出警报。如果没有巨大的人员投入,全世界的数百个副本不可能人工监控。通过消除这一运营任务,系统的伸缩性变得更好,需要的运营支持也更少。

2.4 小结

服务应该包含有利于运营的功能,而不仅仅有利于最终用户。运营人员请求的功能旨在构建稳定、可靠、高性能的服务,这些服务有很好的伸缩性,可以以高成本效益的方式运行。尽管这些功能不是客户直接请求的,更好的运营效率最终会惠及客户。

运营人员需要许多支持日常运营的功能。他们还需要所有运营过程、故障情境及功能控制的完整文档。他们需要身份验证、授权和访问控制机制,以及速率限制功能。运营人员必须能够用开关启用和禁用新功能,这在全局上有利于试运行和回滚,在单个用户上可用于 Beta 测试和增值服务。

选择服务平台

听到有人大肆兜售“云”作为所有计算问题的万能解决方案时，我带着禅僧般的微笑，静静地用“小丑”(Clown)代替“云”(Cloud)。

——Amy Rich

服务在称作平台的计算基础设施上运行。本章概述云计算中的各类平台、它们提供的功能以及优势和劣势。我们不提供具体产品的研究，而是提供一个分类，帮助理解各种服务。选择不同服务的策略在本章最后总结。

“云”这个词是有歧义的；对于不同人来说含义不同，已经因市场上天花乱坠的宣传而失去了意义。我们使用如下具体术语代替：

- ❑ **基础设施即服务 (Infrastructure as a Service, IaaS)**：准备就绪、可供使用的计算机和网络硬件（实际或者虚拟）。
- ❑ **平台即服务 (Platform as a Service, PaaS)**：软件运行于供应商提供的框架或者栈上。
- ❑ **软件即服务 (Software as a Service, SaaS)**：以网站的形式提供的应用程序。

图 3.1 描述了每种服务的典型消费者。SaaS 应用程序是用于最终用户、满足特定细分市场的。PaaS 为开发人员提供平台。IaaS 适用于寻求构建自身平台，在此基础上构建应用，从而提供最大定制性的运营商。

在本章中，我们讨论由第三方供应商提供的这类服务，因为这是一般的情况。

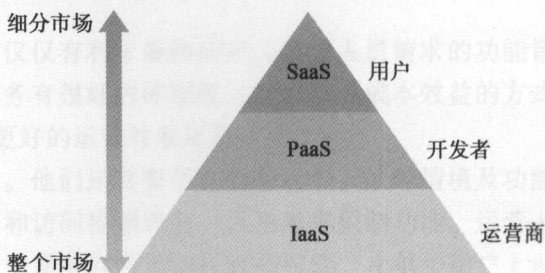


图 3.1 SaaS、PaaS 和 IaaS 消费者

平台可以沿着 3 个轴描述：

- 服务抽象水平：IaaS、PaaS、SaaS
- 机器类型：物理、虚拟或者过程容器
- 资源共享水平：共享或者私有

3.1 服务抽象水平

抽象本质上就是用户与原始机器本身细节的距离。也就是说，你提供的是一台原始的机器（低抽象），还是以高级别 API 的形式提供、封装了需要完成的任务而非完成任务方式（高抽象）的服务？离原始机器越近，就拥有更多的控制。抽象水平越高，就越不需要担心自己构建基础设施的技术细节，越能够专注于应用程序。

3.1.1 基础设施即服务

IaaS 提供裸机，这些裸机连接网络并且为安装操作系统和你自己的软件做好了准备。这种服务提供商提供基础设施，客户可以专注于应用程序本身。

供应商提供的机器通常是虚拟机，但是也可能是物理机器。提供商负责基础设施：机器本身、电源、散热和联网，提供互联网访问以及所有数据中心的运营。

必知术语

服务器：提供函数或者 API 的软件（不是硬件）。

服务：由许多服务器组成的用户可见系统或者产品。

机器：虚拟或者物理机器。

超额预定：提供容量 X 的系统用于需要容量 Y 的场合，其中 $X < Y$ 。用于描述潜在或者实际需求。

预定不足：超额预定的反面。

尽管服务提供商管理其基础设施层次，但是 IaaS 并未能免去所有工作。客户必须花费很多精力协调所有部件、理解和调整各部分使其很好地配合，还要管理操作系统（因为你对 OS 有全面的控制）。

提供商根据计算时间、存储和网络流量收费。这些成本将影响应用程序的架构。将信息保存在本地和通过网络读取相比有不同的成本，会影响设计选择。如果信息经常通过网络访问，网络费用可以通过缓存或者在本地保存更多信息来减少。但是，附加的本地存储自身也有成本。这些设计细节很重要，因为如果不加注意，可能会在月底收到令人瞠目的巨额账单。以上所述都是需要开发及业务团队一起研究的重点。软件和运营选择都有实际的成本和折衷。

提供商的性能特征可能有很大的不同。在比较提供商时，重要的是为本地存储、远程存储、CPU 和网络性能制定基准。有些提供商的远程存储显著地快于其他提供商。在一天的不同时间重复这些基准工作，有些服务提供商在每日的高峰时段可能出现很高的丢包率。为某个提供商做出的设计决策对其他提供商可能不是正确的选择。

在 IaaS 服务中，分区或者“可靠性区域”从地理上分段服务，提供了地区性正常运行时间的保证。虽然尽一切努力确保可靠的服务，但是仍无法避免因维护或者自然灾害等不可抗力引起的停机。服务提供商应该将服务分为多个区域，保证计划内停机不会同时发生在多个区域。每个区域之间的距离应该足够远，使自然灾害不太可能一次袭击多个区域。这使得客户可以在一个区域内保持其服务，在必要时故障切换到另一个区域。

例如，一个服务提供商可以为 4 个区域服务：美国东岸、美国西岸、西欧和东欧。每个区域的构建和管理与其他区域的相关度控制在一定限度内。至少，客户应该将服务布设在一个区域，并制定故障切换到另一个区域的计划。较为成熟的计划应该在每个区域中运行服务，所有位置之间具备负载平衡能力，自动将流量从任何停机的区域切换到其他区域。我们将在第 6 章更详细地介绍。

这种地理上的分散还使得客户能更好地管理服务的延迟。信息的传输需要时间，所以从附近的数据中心提供服务通常更快。例如，服务的架构可能是用户的数据保存在一个区域，备份保存在另一个区域。来自纽约的用户将数据保存在美国东岸区域，备份保存在西欧区域。在停机时，用户从备份区域得到服务。在这种情况下，服务没有那么快，但是至少可以访问数据。

IaaS 提供商已经扩展，不仅仅能够提供简单的机器和网络，有些提供商提供不同的存储选项，包括关系（SQL）和非关系（NoSQL 或者键/值）数据库、高速存储选项和冷存储（廉价但延迟为几小时或者几天的海量数据存储）。更高级的联网选项包括虚拟专用网（Virtual Private Network, VPN）——可访问的私有网络和负载平衡服务。许多提供商同时提供局部负载平衡和全局负载平衡（在第 4 章中将说明）。有些提供商提供弹性伸缩服务，这种服务在需要容量时根据需求自动分配和配置更多机器。

同时提供 IaaS 和 PaaS 的提供商往往提供两者皆可用的高级托管服务，模糊了两者之间的界线。

3.1.2 平台即服务

PaaS 使你可以从供应商提供的框架运行应用程序。这些服务提供高价值，因为它们管理基础设施的所有方面，甚至管理应用程序栈的大部分。它们提供高弹性的伸缩服务，处理额外的负载而不需要任何输入。一般来说，你甚至不知道应用程序专用的具体资源。

例如，在 Google AppEngine 中，你上传应用程序级软件，Google 负责其他工作。框架（平台）自动提供负载平衡和伸缩性。用户越活跃，Google 就为应用程序提供越多机器。在内部，该系统管理带宽、CPU 分配甚至身份验证。你的应用程序可能与数百个其他程序共

享相同机器，也可能需要数百台专用机器的资源。除了限制资源使用以控制成本，没有必要管理这些决策。

PaaS 提供商根据 CPU、带宽和存储的使用量收费。除了收费更高以外与 IaaS 基本类似，这是为了补偿所提供的更全面的框架。

PaaS 的缺点是受限于提供商所提供的平台。这些平台通常是可以编程的，但是不一定可扩展。你不能直接访问操作系统。例如，在供应商将你的二进制代码或者公共库作为服务的一部分之前，可能无法添加它们。进程一般在安全的“监狱”（类似于 UNIX 的“chroot”受限环境）中运行，目标是避免它们挣脱服务框架。例如，某个 PaaS 提供 Python 语言但是没有提供 Python 图像库（PIL）。用户无法安装该库，因为框架不允许包含编译型语言编写部分的 Python 库。

PaaS 提供许多高级服务，包括存储服务、数据库服务及许多与 IaaS 一样的服务。有些提供商提供更高级的服务，如 Google 机器学习服务，该服务可以用于构建推荐引擎。附加的服务定期公布。

3.1.3 软件即服务

SaaS 就是我们以前所称的“网站”，后来市场部门决定采用“即服务”的说法，使其更有吸引力。SaaS 是可以通过 Web 访问的应用程序，这个应用程序就是服务，可以像对任何网站一样与之交互。提供商处理所有硬件、操作系统和平台细节。

常见的例子包括：Salesforce.com，代替本地运行的销售团队管理软件；Google Apps，消除了本地运行电子邮件和日程安排软件的必要；Basecamp，代替本地运行的项目管理软件。在许多公司中常见的业务过程几乎都可以以 SaaS 形式提供：人力资源（HR）的雇用和绩效管理职能；工资表、费用跟踪和总分类账的会计职能；IT 事故、请求和更改管理系统；市场和销售管理的许多方面。

SaaS 的主要卖点是客户不必担心软件安装、升级和运营。不需要下载客户端软件。服务完全由提供商管理、升级和维护。因为服务通过 Web 访问，可以从任何位置使用。

作为 SaaS 提供商，你必须设计服务，进行模糊升级和其他运营细节。开发人员必须避免需要客户端软件或者浏览器插件的功能。在服务的设计中，你必须意识到，因为可以从任何地方访问服务，所以一定会从任何地方访问，包括移动设备。这影响了架构和安全性决策。

必须让客户能够轻松地开始使用你的服务。注册没有必要与销售人员进行对话，而应该可以通过 Web 进行，可能需要提交信用卡或者其他支付信息。如果每个用户都必须首先与客户服务代表对话才能安排账户的创建，Facebook 也不可能发展到现在的规模。数据的导入和功能的启用也应该是自助服务。如果数据导入或者其他操作必须与客户支持人员一起才能进行，Salesforce.com 之类的产品也不可能以这样的速度发展。

人们能够以自助服务的方式离开服务也很重要。这意味着，即使该方式会使用户转投

竞争对手更加容易，用户也应该能够导出或者检索他们的数据，并关闭账户。客户对合约到期或者感到不满意时迁移到其他应用程序的能力很关心。我们相信，通过使用户不可能或者难以导出数据，将他们锁定在某种产品上的行为是不道德的。这种做法（称作供应商锁定）应该被视为产品不值得信任的“危险信号”。

许多 SaaS 服务频繁升级（往往没有警告），提供的培训机会很少。用户应该可以为了规划、培训和用户验收测试等目的访问重要的新发行版本。为用户提供选择转移到重要发行版本的日期的机制，或者提供两条道路：对希望毫无延迟地得到新功能的客户提供“快速发行”，对希望按照公布的时间表（通常在快速发行的 2~3 周之后）获得新功能的客户提供“计划发行”。

最后，数据隐私和应用程序托管策略与客户可能存在冲突。隐私策略必须是所有客户隐私策略的超集。你可能需要为某些客户提供更高的安全性，可以将他们与其他客户分开。

3.2 机器的类型

运行服务的机器类型有 3 种选择：物理机器、虚拟机和过程容器。选择机器类型是一个技术决策，每种类型的机器都有不同的性能、资源效率和隔离能力。应该按照所需的技术属性决定使用的类型。

IaaS 通常提供最广泛的选择。PaaS 一般模糊了所使用的机器类型，因为用户工作于隐藏这种区别的框架之中。也就是说，大部分 PaaS 提供商使用容器。

3.2.1 物理机器

物理机器（physical machine）是具备一个或者多个 CPU 以及内存、磁盘和网络子系统的传统计算机。这些资源受控于操作系统，操作系统的任务是作为协调共享这些资源的进程的“交通警”。分配给运行进程（在系统上运行的一个程序）的资源是实际硬件资源。因此，它们的性能相对容易预测。

3.2.2 虚拟机

虚拟机（virtual machine）在物理机器被分区、在每个分区运行单独操作系统的时候创建。运行于虚拟机上的进程很少意识到（或者完全没有意识到）它们不在物理机器上运行。它们不能访问同一个物理机器上运行的其他虚拟机的资源（如磁盘或者内存）。

虚拟机使计算更加高效。当今的物理机器非常快速和强大，有些应用程序不需要单一机器的全部资源。超过的容量被称作搁置容量（stranded capacity），因为在当前的形势下它无法使用。在许多较小的虚拟机之间共享大型物理机器的能力，可以创建规模与其需求相适应的虚拟机器，有助于减少搁置容量。

在同一台机器上运行多个服务器也能够减少搁置容量。但是，虚拟化提供比简单多任

务更好的隔离。

例如,当两个应用程序共享一台机器,一个应用程序超载或者发生导致消耗大量 CPU、磁盘空间或者内存的问题时,会影响另一个应用程序的性能。现在假定这两个程序运行在自己的虚拟机上,每个机器分配一定量的 CPU、磁盘空间和内存。这种布局使每个应用程序更好地与其他应用程序发生的问题隔离。

有时候,使用虚拟机是组织上的原因。组织中的不同部门可能彼此没有足够的信任或者充足的跨部门记账选项,无法在同一台机器上运行软件。但是,如果每个部门能够创建自己的虚拟机,就可以共享一个物理机器池。

有时候,使用虚拟机的原因是后勤。在一台机器上运行 5 个服务要求任何操作系统补丁或者升级须得到 5 个服务的批准。如果每个服务运行于自己的虚拟机,则不同服务的更新和补丁可以按照不同的计划安排进行。在这些情况下,虚拟机可以实现操作系统级的隔离。

1. 虚拟机的好处

虚拟机的创建和删除很快。请求虚拟机与虚拟机可用之间的前置时间很短。有些系统可以在低于一分钟的时间内启动新的虚拟机。因此,很容易为某个特殊任务创建虚拟机并在任务完成时删除。这种虚拟机称作**短寿机器**(ephemeral machine)。有些系统在许多物理机器上创建数百个短寿机器,进行并行运算工作,然后在工作完成时删除这些机器。

因为虚拟机通过软件控制,所以虚拟化系统是可编程的。可以使用 API 创建、启动、停止、修改和删除虚拟机。可通过编写软件大规模地编排上述功能。这对于物理机器是不可能的,物理机器必须通过人工上架、连线和配置。

虚拟机的功能由现代 CPU 中的虚拟化支持结合**虚拟机监控器**(VMM)提供。现代 CPU 有助于分割内存和 CPU 时间以创建虚拟机的特殊功能。磁盘、网络和其他 I/O 设备的访问由芯片级或者设备级的仿真处理。

有些虚拟化系统允许虚拟机在物理机器之间移动。正如笔记本电脑可以进入休眠并在以后唤醒一样,VMM 使机器进入休眠状态,将内存和所有其他状态复制到不同的物理机器,在那里继续活动。可以对这一过程加以协调,提前进行大部分复制,使机器“冻结”的时间短于 1 秒。这就使得当前物理机器需要升级或者维修时,可以将虚拟机移动到不同的物理机器,或者在计划维修停机之前移动到不同的故障域。

2. 虚拟环境中的 I/O

硬件虚拟机(Hardware Virtual Machine, HVM)在芯片级执行 I/O 仿真。使用 HVM 时,虚拟机的操作系统认为真安装了某种设备,如 SATA 硬盘控制器,这使虚拟机可以使用未经修改的操作系统。但是,这种仿真相当缓慢。在 HVM 上,每当 OS 试图访问 SATA 控制器,CPU 的虚拟化功能检测到这种访问,停止虚拟机,将控制交给 VMM。VMM 执行磁盘请求,仿真真正的 SATA 控制器,并置入真实芯片得到的结果。然后,虚拟机可以从停止

的地方继续，在一无所知的情况下看到结果。

半虚拟化 (Paravirtualization, PV) 在设备级执行 I/O 模拟。PV 要求修改操作系统，使正常执行的 I/O 调用由对 VMM 的请求代替。VMM 处理 I/O 并返回结果。这种修改通常采取设备驱动（看上去是标准硬盘、显示器、键盘等）的形式，但是实际上是与 VMM 通信。PV 可以更高效地执行请求，因为它在更高级别的抽象上捕捉请求。

虚拟机从物理机器分配固定数量的磁盘空间、内存和 CPU。VM 所看到的硬盘实际上可能是物理机器上的一个大文件。

3. 虚拟机的缺点

有些资源（如 CPU 内核）是共享的。假定一台物理机器有一个 4 核 CPU。可以为一个虚拟机器分配 3 个虚拟内核，同时为另一个虚拟机分配 3 个虚拟内核。VMM 将在 4 个物理核心上共享 6 个虚拟核心的负载。可能有些时候，一个虚拟机相对空闲，并不需要所分配的全部 3 个虚拟核心。但是，如果两个虚拟机都大负载运行，需要全部 6 个虚拟核心，则每个核心只能得到 CPU 的一部分注意力，所以会运行得较慢。

虚拟机能够检测 CPU 争用。在 Linux 和 Xen 虚拟化管理器上，这被称为“窃取时间”：虚拟机的 CPU 时间因为分配给其他虚拟机而丢失了 (Haynes, 2013)。IaaS 提供商通常不能保证存在多少窃取时间的情况，也不能提供控制机制。Netflix 发现处理这种情况的唯一手段是保守。如果在 Amazon Web Services (AWS) 的虚拟机中检测到很多“窃取时间”的情况，Netflix 将删除虚拟机并重新创建。如果该公司很幸运，新虚拟机将在超额预定情况较少的物理机上创建。这是令人遗憾的情况 (Link, 2013)。

有些资源以不受限的方式共享。例如，如果一个虚拟机产生巨大的网络通信量，其他虚拟机可能受到不利的影响。这也是磁盘 I/O 的典型情况，硬盘每秒可能执行如此之多的磁盘 I/O，该数量受限于从计算机到磁盘的带宽。在磁盘 I/O 带宽等资源短缺时，这种情况称为资源争用。

虚拟机是非常重量级的，它们运行完整的操作系统，需要许多磁盘空间。即使不使用，它们也会保留分配的内存，底层 OS 无法将这些内存重新分配给其他机器。因为虚拟机运行的是完整的操作系统，其运营负担类似于完整的机器，需要监控、补丁、升级等。而且因为运行的是完整的操作系统，每个操作系统还运行许多后台服务进程，如维护任务和服务守护进程。这些进程占据资源，给系统管理团队带来运营负担。

3.2.3 容器

容器 (container) 是一组运行于操作系统、与其他同类进程组隔离的进程。每个容器有一个环境，具备自己的进程命名空间、网络配置和其他资源。这些进程拥有权限的文件系统由主机上的一个子目录组成。特定容器中的进程将该子目录视为自己的根目录，如果主机没有特殊的措施，则无法访问该子目录（及其子目录）之外的文件。这些进程都运行于相同的操作系统或内核上。因此，你无法像对虚拟机那样，让一些进程运行于 Linux 下，其

他进程运行于 Windows 下。

与分配大块 RAM 和磁盘的虚拟机不同，容器消费资源的粒度和进程相同，因此浪费较少。

容器中的进程作为一个组受控。如果容器被配置为有内存限制，该容器中所有进程使用的内存总量不能超过该限值。如果容器分配了固定的磁盘带宽，容器中的进程作为整体实施该限制。Solaris 的容器称作“分区”(Zone)，可以分配网卡和规定的网络带宽，以控制带宽资源争用。Linux 上的容器可以分配不同数量的磁盘缓存，使一个容器的缓冲区“颠簸”不会影响到另一个容器的缓冲区。

容器中的进程以其他方式隔离。容器只能杀死在其容器中的进程或者与之交互。相反，不在容器中的进程可以杀死所有进程（即使它们在单独的容器中）或者与之交互。例如，Shell 命令 `ps` 在 FreeBSD 容器（称作“监狱”）运行时，只显示在该容器中运行的进程。这不是一个恶作剧。容器对其他进程没有可见性。然而，同一个命令在主机上从任何容器之外运行时，它会显示所有进程，包括每个容器中的进程。因此，如果你登录主机（不是特定的容器），就有全局可见性，可以作为所有容器的管理员。

每个容器都有自己的软件包、共享库和需要的其他支持文件的拷贝。运行在同一台机器上的两个容器不可能有相互依赖性或者版本冲突。例如，如果没有容器，一个程序可能需要某个库的特定版本，而另一个需要完全不同的版本，无法使用安装的其他版本。这种“依赖性地狱”很常见，但是当每个程序被放入不同容器时，它们可以有自己的库版本，从而避免这种冲突。

容器非常轻量，因为它们不需要完整的 OS。只有软件所需的特定系统文件被复制到容器中，系统在文件需要的时候分配磁盘空间，而不是预先分配大的虚拟磁盘。容器运行的进程较少，因为它只需要运行与软件相关的进程。SSH 等系统后台进程和其他守护进程不在容器中运行，因为它们存在于外部操作系统。使用虚拟机时，每个机器都有这些完整的守护进程。

容器不同于虚拟机。每个虚拟机是一个黑盒子。登录到物理机器的管理员（如果不使用技巧）无法窥探单独的虚拟机。虚拟机可以运行与物理主机不同的操作系统，因为它仿真整台机器。虚拟机是更大规模、粒度更粗的资源分配方式。当虚拟机启动时，分配一定数量的 RAM 和磁盘空间，供虚拟机专用。如果虚拟机不使用所有的 RAM，这些剩余的 RAM 也无法供其他人使用。虚拟磁盘往往难以改变尺寸，所以人们创建大于需求的磁盘，降低容器需要增大的可能性。这些额外容量无法供其他虚拟机使用。这种情况称作“搁置资源”。

容器和虚拟机有一些相同的缺点。主机宕机会影响所有容器。这意味着为主机打补丁的计划停机时间和计划外的停机都会影响所有容器。但是主机所要做的工作不多，所以可以运行精简版的操作系统，需要的补丁和维护较少。

迄今为止，我们已经讨论了容器的技术特征，但是用容器所能完成的工作更加激动

人心。

容器通常是 PaaS 中的底层技术。它们使客户可以相互隔离，同时仍然共享物理机器。因为容器消耗的只是当时所需的资源，它们还是提供这类共享服务的更高效手段。

Docker 等系统为软件定义标准化容器，人们可以不将软件作为一个包分发，而是分发一个容器，包含软件及运行所需的所有条件。这个容器可以创建一次，在许多系统上运行。

容器是自包含的，消除了依赖和冲突。我们不需要分发软件包和一系列其他相关软件包和系统需求，所需要的就是标准化的容器和支持该标准的系统。这大大简化了软件的创建、存储、交付和分发。由于许多容器可以共存于相同的机器上，最后这台机器就像一个大旅馆，可以为许多客户提供服务，即使他们都有独特的需求，也能够以同样的方式对待。

标准化集装箱

行业显著改善过程的常见方法之一是标准化交付机制。引入标准化集装箱给货运行业带来了革命性的变化。

过去，在船上装卸单独货物通常依靠手工。每件货物的尺寸和外形都不相同，所以必须以不同的方式搬运。

标准化集装箱造就了运送产品的全新方式。因为每个集装箱的形状和尺寸相同，装卸工作可以更快完成。单个集装箱可以容纳许多单独货物，但是因为它们作为一组运输，改变货物的运输模式很快速。海关可以检验特定集装箱中的所有货物并贴上封条，只要封条没有损坏，集装箱运送路径上的其余关口就无需再检查。

随着其他运输模式采用标准集装箱，产生了联合运输的概念。集装箱可以在工厂中装载，不管在卡车、列车或者船只上都作为一个单元。

这一切始于 1956 年 4 月，当时 Malcom McLean 的 SeaLand 公司组织了第一批使用标准化集装箱的货物，从新泽西（Tom 的家乡）运往得克萨斯（Levinson，2008）。

3.3 资源共享水平

在“公共云”中，第三方拥有基础设施，并用其为许多客户提供服务。共享可能是细粒度的，不同客户的处理和数据混合在同一机器上。共享也可以采用更细致的分段方法，就像公寓楼中的租户之间有精细定义的分区一样。在“私有云”中，公司在自己的场地内运行自己的计算基础设施。这些设施可能是为了专门的内部项目而建立的，或者（更常见的情况）作为内部服务提供者为公司内的项目和部门服务。也可以创建混合云，例如在租赁的数据中心空间内运营私有云。

私有或者公用平台的选择是一个业务决策，基于 4 个因素：依从性、隐私、成本和控制。

3.3.1 依从性

根据业务、大小、位置和公有或者私有地位，公司受到不同法规的监管。它们与所有适用监管法规相关的依从性可以被审计，未能通过审计可能带来严重的后果，例如公司在通过下次审计之前无法开展业务。

对某些数据或者服务使用公共云可能造成公司无法通过依从性审计。例如，欧盟数据保护法令规定关于欧盟公民的某些数据不能离开欧盟。除非公共云提供商有足够的控制能够确保上述情况不会发生（甚至在故障切换中），否则将数据移入公共云的公司将无法通过审计。

3.3.2 隐私

使用公共云意味着数据和代码保存在其他人的设备和设施上。他们可能不能直接访问你的数据，但是有可能在你不知情的情况下获得访问权。好奇的员工（可能有恶意企图，也可能没有）可能使用可查看数据的诊断工具浏览。服务提供商处置旧设备时如果没有正确地擦除存储系统，数据可能会在无意中泄露。

因为这些风险，服务提供商应该在合同中说明如何维护你的数据。除了合同之外，供应商们知道，如果他们想保留客户，就必须赢得信任。他们通过策略的透明性和外部审计机构对其遵守规则的验证来维护这种信任。

公共云的另一个问题是如何处理执法机关的请求。如果执法机关的官员有正当理由访问数据，他们可能让第三方在未告知你的情况下提供访问权。相反，在私有云中，他们访问数据的唯一渠道就是让你知道他们的请求（即使在自己的场所中，秘密技术仍然可能是隐蔽的）。

你的数据可能意外地曝光。由于软件缺陷、员工错误或者其他问题，你的数据可能暴露给其他客户或者全世界。在私有云中，其他客户都来自同一家公司，这种风险也许是可以接受的，这种事故可能得到包容，不会为公众所知。在公有云中，数据可能暴露给任何人，甚至可能是你的竞争对手，这可能成为头条新闻。

3.3.3 成本

使用公共云的成本可能低于（也可能不低于）自行构建必要基础设施的成本。构建这样的基础设施需要大量工程人才——从数据中心散热系统、电气设备维护和设计方面的物理工程，到运营数据中心并提供服务的技术专业人士。所有这些都可能很昂贵。将这笔费用分摊到许多客户可以降低成本。相比之下，自行构建能够节约成本是因为垂直整合（vertical integration）的好处。垂直整合意味着通过自行处理所有级别的服务交付、消除“中间人”和服务提供商利润带来的成本，从而降低成本。存在一个平衡点，在该点上垂直整合变得更为经济。计算总拥有成本（TCO）和投资回报率（ROI）有助于确定哪一种选项是

最佳方案。

确定私有云和公共云的成本哪一个更合适，类似于决定你是租房还是自购房屋居住。从长期看，拥有房屋比租赁更便宜。但是，在短期租赁可能更便宜。租赁一个旅馆房间过夜比在城市中购买房屋，然后在次日售出更有意义。

类似地，如果使用云的全部空间，且使用的时间足以带来相应的回报，那么构建私有云是有意义的。如果需求很小或者是短期需求，那么使用第三方提供商更有意义。

3.3.4 控制

私有云提供更多的控制。你可以精确指定将要使用的硬件种类、建立的网络拓扑等。对于所需要的任何功能，关键是自行创建、购买还是获得许可证？发生更改的快慢取决于组织的能力。在公共云中，你的控制权较弱，必须从有限的功能集中选择，虽然大部分提供商对功能申请的反应很灵敏，但是你并不是唯一的客户。提供商必须将重点放在许多客户使用的功能上，可能无法为你提供所需的专门功能。

让供应商负责所有硬件选择，意味着失去了规定低级硬件需求（特定 CPU 类型或者存储产品）的能力。

托管服务提供商的合同问题

你所签署的合同是对提供商的预期及其对你（客户）所承担义务的基准。下面是向潜在提供商提出的一些关键问题：

- 1) 如果你打算退出合同，能否带走所有数据？
- 2) 使用物理机器时，如果你希望离开，可否购买该机器？
- 3) 如果供应商破产，对你的服务器和数据会发生什么情况？它们是否与破产程序捆绑？
- 4) 互联网带宽由供应商提供，还是必须自行安排？如果由供应商提供，你所连接的是哪个互联网服务提供商（ISP），完成了多少超额订购？采用何种硬件和对等传输冗余性？
- 5) 是否执行备份？如果有，备份的频率如何，采用何种轮换策略？你可以申请访问备份，还是它只能供供应商在紧急情况下使用？恢复测试的频率如何？
- 6) 供应商怎样保证其服务水平协议（Service Level Agreement, SLA）中所列的容量和带宽？如果违反 SLA，是否可以退款？

3.4 主机托管

主机托管（Colocation）虽然没有被特别地视为“云环境”，但是它是提供服务的实用手段。主机托管发生在数据中心所有者将空间租赁给其他人（租户）时。租赁数据中心空间对

于中小企业（甚至大型企业）非常经济。构建自己的数据中心是一项巨大的投资，需要有关散热、设计、网络、位置选择、物业管理等的专业知识。

“主机托管”这一术语来自于电信界。过去，电信公司是少数构建数据中心的公司之一，数据中心用于容纳它们的设备和系统。有些第三方公司向电信公司的客户提供服务，如果它们将自己的设备放在电信公司的数据中心，会更容易实现。因此，他们将自己的设备和电信公司的设备放在一起。近年来，任何数据中心空间租赁都被称作主机托管服务。

当你需要小的或者中等的数据中心空间时，这种服务很理想。这种数据中心通常设计精良、运营得当。获得空间构建数据中心可能花费数年的时间，而使用主机托管设施很快就能正常运营。

需要许多小空间时，主机托管也很有用。例如，某公司可能想要在 12 个国家里各有一个机架设备，为这些国家中的客户提供更好的访问。

ISP 往往将扩展其网络，进入主机服务提供商领域，以便租户直接连接到它们。直接连接到 ISP 改善了 ISP 用户的访问速度。另外，主机托管提供商可以向租户提供互联网访问服务，这种访问往往融合了两个或者更多 ISP 的连接。租户们可以利用这些能力，而不需要管理自己的 ISP 连接和关系。

3.5 选择策略

选择 IaaS、PaaS 和 SaaS 时有许多策略可以使用。

下面是一些我们使用过的策略：

- ❑ **默认为虚拟：**尽可能使用容器或者虚拟机，仅在其他方法无法实现性能目标时选用物理机器。例如，物理机器通常有更好的磁盘 I/O 和网络 I/O 性能。常常见到除了特定的高 I/O 要求的数据库之外，全部采用虚拟化环境的情况。Web 负载平衡器也往往使用物理硬件，因为这样可以利用高带宽网卡或者从专用网络连接获得可预测性能的好处。

- ❑ **根据成本做出决策：**根据成本选择公共云或者私有云。建立一个业务计划，描述在内部、私有基础设施和使用公共云提供商相比较的全部项目成本。选择成本较低的选项，对于多年的项目，自行完成可能较为便宜。对于短期项目，公共云可能是更经济的选项。

- ❑ **利用提供商的专业能力：**利用云提供商在创建基础设施方面的专业知识，使你的员工可以专注于应用。这种策略对于小型公司和创业公司特别有吸引力。如果只有一两名开发人员，在可以利用公共云服务时构建大量的基础设施是不可想象的。公共云的运营通常比私有云更专业，因为提供商必须与其他提供商争夺客户。当然，私有云可能运营得很专业，但是小规模单位难以做到，尤其是在没有专职人员的情况下。

❑ **快速启动：**利用云提供商，尽可能快速地投产。最大的成本往往是“机会成本”：如果因为动作过于缓慢而失去机会，节约多少钱都没有意义。与公共云提供商签约可能比自行建设更昂贵，但是自行构建基础设施可能需要花费几个月甚至数年，到那时机会早已消失。当产品成功与否尚不确定时，基础设施建设也可能是浪费时间。在产品生命期的初级阶段，整个思路都可能是实验性的，包括快速尝试新事物并反复迭代以尝试新功能。使用公共云服务，能够向早期测试者快速提供最小化的产品，了解产品是否可行。如果产品可行，可以考虑私有基础设施。如果产品不可行，就不需要浪费时间构建基础设施。

❑ **实施短生命期计算：**对于短期项目使用短生命期计算。短生命期计算需要在短期内建立大规模的计算基础设施。例如，想象一家公司发动一场广告宣传活动，在几周时间内将数百万用户吸引到其网站，而在事后访问量急剧下降。利用公共云提供商，该公司可以短暂地扩展到数千台机器，然后在不需要时去掉这些机器。这常常用于大型数据处理和分析项目、概念验证项目、生物科技数据挖掘和意外网站流量爆发的处理。构建大规模的基础设施用于这么短的时间是不合理的，但是云服务提供商的专长就是提供这种计算设施。总体而言，利用率将是平滑的，负载是恒定的。

❑ **利用云提供额外的容量：**确定基准容量需求并在内部建设，然后使用云提供超出正常的容量。这往往比内部容量建设或者专门使用公共云提供商更有成本效益。

❑ **利用优越的基础设施：**通过优越的基础设施获得优势。在这种策略中，利用创建定制化基础设施的能力，获得竞争优势。这可能包括构建自己的数据中心以控制成本及资源利用率，或者选择 IaaS 代替 PaaS，利用操作系统和软件级的定制能力。

❑ **开发内部服务提供者：**建立内部服务提供者，以控制成本和维护隐私。计算基础设施往往在规模很大时才具备成本效益，这就是公共云提供商能够如此经济地提供服务的原因。但是大型公司可以构建许多内部客户共享的大型基础设施，实现类似的规模经济。因为这种云是内部使用的，因此是私有云——这往往是高度监管下的行业所必需的。

❑ **签署外部运行的场内服务合同：**有些公司为你运营内部云服务。它们的不同之处在于提供的控制和定制程度。你可以从它们的专业知识中获益，并且因为对设备的所有权和控制权而缓解隐私问题。

❑ **最大化硬件产出：**通过回避虚拟化，追求从计算机中挤出生产率和效率的战略。当基础设施中有数十万台计算机、几百万个核心的时候，效率提高 1% 等同于获得数千台新计算机。因为虚拟化损失的效率可能造成巨大的开支。在这种策略中，使用物理机器而非虚拟机，服务十分紧凑以最大化利用率。

❑ **实施裸机云：**像虚拟机云一样管理物理机器基础设施。通过配给虚拟机所用的同一个 API 提供物理机器。利用一些规划，启用虚拟机的好处可能适用于物理机器。有

些公司并不为每台物理机器选择完全定制的配置，而是购买数百或者数千台机器，全部以相同方式配置和管理，作为为部门或者个人保留的机器池。它们提供用于分配机器、清理和重装 OS、重启、访问控制和归还到机器池的 API 来实现上述功能。分配可能不像虚拟机那么快速或者动态，但是可以实现许多相同的好处。

3.6 小结

本章研究了一些平台。基础设施即服务（IaaS）为你的 OS 和应用程序安装提供物理或者虚拟机器。平台即服务（PaaS）提供 OS 和应用程序栈或者框架。软件即服务（SaaS）是基于 Web 的应用程序。

可以用物理或者虚拟服务器创建自己的云，自行托管机器或者利用 IaaS 提供商。具体的业务需求将指导你确定组织的最佳行动步骤。对于任何指定的服务或者应用程序，构建健全可靠的架构都有着广泛的选择空间。在下一章，我们将研究各种架构。

练习

1. 根据成本、可配置性和控制，比较 IaaS、PaaS 和 SaaS。
2. 采用软件即服务时需要注意哪些事项？
3. 列出虚拟机的关键优势。
4. 为什么选择物理机器而不选择虚拟机？
5. 哪些因素会使你选择私有云而非公共云服务？
6. 你目前的组织使用哪种选择策略？使用这种策略有何好处和注意事项？

应用程序架构

模式是某个语境中反复出现的问题的解决方案。

——Christopher Alexander

本章研究设计应用程序和其他服务时的构件。前一章讨论了云平台选项，现在我们上一层楼，进入应用程序架构。

我们首先研究常见的 Web 服务架构，从单个 Web 服务器到多机设计，逐渐增长，直到设计适合于大型的全球性服务。然后，我们研究 Web 应用程序常见的后台架构：消息总线 and 面向服务架构。

本章的大部分示例假定服务是使用超文本传输协议（HTTP）、基于 Web 的应用程序。用户运行 Firefox、Chrome 或 Internet Explorer 等 Web 浏览器，按照 HTTP 术语，这称为客户端。每个网页请求包括用 HTTP 协议和 Web 服务器（通常运行于互联网另一处的机器上）通信。Web 服务器作为 HTTP 协议的服务器端，接受 HTTP 连接，解析请求并进行处理，生成响应。响应是发送给客户端的 HTML 页面或者其他文件。然后，客户端为用户显示网页或者文件。通常，每个 HTTP 请求或者查询（query）是单独的 TCP/IP 连接，但是对协议的一些扩展可以让一个会话处理许多 HTTP 请求。

有些应用程序使用 HTTP 之外的协议，例如它们可能实现自己的协议。有些非 Web 应用程序使用 HTTP，例如，手机应用可能使用 HTTP 与 API 通信，发起请求或者收集信息。虽然我们的大部分示例假定 Web 浏览器使用 HTTP 协议通信，但是这些原则适用于任何客户 / 服务器应用程序和协议。

4.1 单机 Web 服务器

我们所要研究的第一种设计模式是用于提供 Web 服务器的单台自给型机器（图 4.1）。该机器运行使用 HTTP 协议通信的软件、接受请求、处理请求、生成结果并发送响应。许多典型的小网站和基于 Web 应用程序采用这种架构。

Web 服务器从 3 种不同的来源生成网页：

- **静态内容**：从本地存储读取文件，不做更改地发送给用户。这些文件可能是 HTML 页面、图像、音乐和视频等内容，或者可下载的软件。
- **动态内容**：在 Web 服务器上运行的程序生成 HTML 以及发送给用户的其他输出。它们可能独立生成，或者以从用户接收的输入为基础生成。
- **数据库驱动动态内容**：这是动态内容的特例，运行于 Web 服务器上的程序查询数据库获取信息，并用这些信息生成网页。在这种架构中，数据库软件与数据在同一台机器上作为 Web 服务器。

并不是所有 Web 服务器都有全部 3 种流量。静态 Web 服务器没有任何动态内容，动态内容服务器可能需要数据库，也可能不需要。单机 Web 服务器是网站和应用中非常常见的配置。对于许多小型应用，这种架构已经足够了，但是有一些局限性。例如，它不能存储或者访问超过一台机器容量的数据，可服务的并发用户的数量也受到机器 CPU、内存和 I/O 容量的限制。

这种系统的可靠性也只能达到单机所能达到的水平。如果这台机器崩溃或者死机，Web 服务在机器维修好之前是停止的。在机器上进行维护也很困难。软件更新、内容更改等操作都可能需要停机时间。

随着机器接收的流量增长，单机 Web 服务器可能超载。我们可以添加更多内存、磁盘和 CPU，但是最终会达到机器的限制。这些限制可能是约束硬件物理连接（物理插槽和端口数量）的设计限制，也可能是磁盘和内存内部互联带宽等内部限制。我们可以购买更强大的机器，但是最终还是会达到限制。随着流量增长，系统不可避免地会达到极限，唯一的解决方案是采用不同的架构。

另一个问题与缓冲区“颠簸”有关。现代操作系统将所有空闲内存用作磁盘缓存，这改善了磁盘 I/O 性能。操作系统可能使用许多不同的算法调整磁盘缓存，这些算法都和确定其他进程或者新的磁盘数据块需要内存时抛弃哪些数据块有关。例如，如果一台机器运行 Web 服务器，OS 将自行调整 Web 服务器的内存占用。如果机器运行数据库服务器，OS 将以不同方式调整，甚至可能选择完全不同的数据块替换算法。

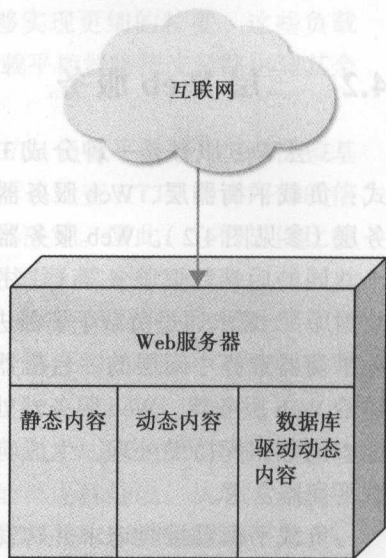


图 4.1 单机 Web 服务架构

问题是，如果一台机器同时运行 Web 服务器和数据库服务器，内存占用可能很复杂，操作系统无法自行调整，它可能选择一种仅对某个应用最优的算法，或者放弃最优算法、选择对所有应用程序都不是最优的默认方案。改善这一状况的尝试包括 Linux 的“容器”系统，如 LXC 和 Lmctfy。

4.2 三层 Web 服务

3 层 Web 服务是一种分成 3 层构建的模式：负载均衡器层、Web 服务器层和数据服务层（参见图 4.2）。Web 服务器都依赖于一个共同的后端数据服务器（往往是一个 SQL 数据库）。请求通过负载均衡器进入系统。负载均衡器选择中间层的一台机器，将请求传递给 Web 服务器。Web 服务器处理请求，可能查询数据库协助处理，生成响应并通过负载均衡器发送。

负载均衡器接收请求并转发给许多副本中的一个——经过配置可以服务同一个 URL 的 Web 服务器。用户和负载均衡器通信，就像它就是 Web 服务器一样。他们没有意识到，这只是许多副本的前端。

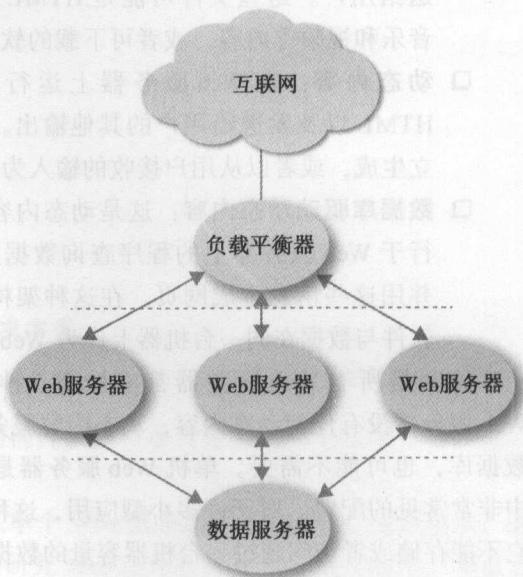


图 4.2 3 层 Web 服务器架构

4.2.1 负载均衡器种类

创建负载均衡器有许多种方法，通常分为 3 类：

- ❑ **DNS 循环（Round Robin）**：这种方法的工作原理是在 Web 服务器名称的 DNS 条目中列出所有副本的 IP 地址。Web 浏览器接收所有 IP 地址，但是首先随机选取其中一个尝试。因此，当多个 Web 浏览器访问网站时，负载将在副本中几乎均匀地分布。这种技术的好处是容易实现且免费，除了 DNS 服务器（已经存在）之外，不涉及任何其他硬件。但是这种技术很少使用，因为它的效果不是很好，且难以控制。这种方式的反应不灵敏，如果一个副本意外死机，客户端将继续尝试访问它，因为客户端缓存 DNS。在 DNS 缓存过期之前，该网站看上去就像下线了一样。对于哪个服务器接收哪些请求没有什么控制，当一个副本出现不寻常的超载情况时，没有简单的方法减少发送给它的流量。
- ❑ **第 3 层和第 4 层负载均衡器**：L3 和 L4 负载均衡器接收每个 TCP 会话，并将其重定位到其中一个副本。会话的每个数据包首先通过负载均衡器，然后到达副

本。响应数据包从副本通过负载均衡器返回。这一模式的名称来自 ISO 协议栈定义：第 3 层是网络（IP）层，第 4 层是会话（TCP）层。L3 负载均衡器根据源和目标 IP 地址（即网络层）跟踪 TCP 会话。不管它已经生成的 TCP 会话数量如何，所有来自指定源地址的流量都发送给同一台服务器。L4 负载均衡器除了 IP 地址之外，还跟踪源和目标端口（即会话层），这能够实现更细的粒度。这些负载均衡器的好处是简单而快速。而且如果副本停机，负载均衡器将把流量路由到其余副本。

□ **7 层负载均衡器**：L7 负载均衡器的工作方式类似于 L3/L4 负载均衡器，但是决策基于协议栈中应用层（第 7 层）中检查到的信息。它们可以检查 HTTP 协议内部的信息（Cookie、首标、URL 等），并根据找到的信息做出决策。因此，它们提供比前几种负载均衡器更丰富的功能组合。例如，L7 负载均衡器可以检查 Cookie 是否发出，并根据该条件向不同组服务器发送流量。这就是一些公司以不同方式处理登录用户的原理。有些公司在最重要的客户登录时设置一个特殊的 Cookie，并配置负载均衡器检测该 Cookie，将流量发送到特别快的服务器。

有些负载均衡器是透明的：请求的源 IP 地址不做修改。但是，大部分情况不是这样的：后端看到的每个请求的源 IP 地址是负载均衡器自身的 IP 地址。也就是说，从后端的角度看，似乎所有请求都来自于一个来源——负载均衡器，请求的实际源 IP 被掩盖了。

当所有请求似乎都来自于同一个 IP 地址时，调试和日志分析即使不能说完全不可能，至少也是令人困惑的。处理这种问题的通常方式是负载均衡器注入一个附加的首标，表示原始请求者的 IP 地址。后端可以在必要时访问该信息。这个首标称作 X-Forwarded-For。第一个代理或者负载均衡器添加该首标、原始客户端 IP 地址和自己的 IP 地址。其他代理和负载均衡器附加自身的 IP 地址。虽然 Web 服务器通常会将客户端 IP 地址记入日志，但是应用程序也可以访问所有中间 IP 地址。

4.2.2 负载均衡方法

对于每个请求，L3、L4 或者 L7 负载均衡器决定将其发送到哪一个后端。这一决策有不同的算法：

□ **循环（Round Robin, RR）**：机器循环轮转，如果有 3 个副本，循环的方式如 A-B-C-A-B-C。故障机器被跳过。

□ **加权 RR**：这种方案类似于 RR，但是将更多查询导向具有更大容量的后端。权重（weight）通常人工配置到每个后端。例如如果有 3 个后端，其中两个容量相同，第三个容量更大，可以处理两倍的流量，循环模式将为 A-C-B-C。

□ **最小负载（Least Loaded, LL）**：负载均衡器从每个后端接收负载信息显示负载的情况。入站请求总是导向负载最小的后端。

□ **带慢启动的最小负载**：这种方案类似于 LL，但是在新后端上线时不会立刻用查询

淹没它，而是从接收低流量开始慢慢积累，直到接收合适的流量。这种方案解决了 1.3.1 节中描述的 LL 问题。

❑ **利用率限制**：每个服务器估算自身可以处理的 QPS 数量，并将其传达给负载均衡器。估算可以根据当前吞吐率或者综合负载测试收集的数据进行。

❑ **延迟**：负载均衡器根据当前请求的延迟停止向某个后端发送请求。例如，当请求所花费的时间超过 100 毫秒，负载均衡器认为该后端已经超载。这种技术能够控制慢速请求的爆发或者病态超载的情况。

❑ **级联**：第一个副本接收所有请求，直到到达其容量限制。任何超出的流量被导向下一个副本，以此类推。在这种情况下，负载均衡器必须准确地知道每个副本所能处理的流量，通常根据综合负载测试的静态配置进行。

4.2.3 共享状态的负载均衡

在许多副本中平衡负载的另一个问题是共享状态。假定一个 HTTP 请求生成下一个 HTTP 请求所需的信息。单个 Web 服务器可以将该信息保存在本地，在第二个 HTTP 请求到达时使用。但是，如果负载均衡器将下一个 HTTP 请求发送给不同的后端会发生什么？该后端没有这些信息（状态），这可能造成混乱。

考虑经常遇到的一种情况：一个 HTTP 请求获得用户名和密码并验证，让用户登录。服务器保存用户已经登录的事实，并从数据库读取其档案，这些信息保存在本地供快速访问。未来对相同机器的 HTTP 请求知道该用户已经登录并且手上已经有了用户档案，所以没有必要访问数据库。

如果负载均衡器将下一个 HTTP 请求发送到不同的后端会怎么样？这个后端不知道用户已经登录，将要求其再次登录，这会令用户恼怒，并且给数据库带来额外的工作负载。

处理这种情况有几种策略：

❑ **粘性连接**：负载均衡器有“粘性”特征，这意味着如果用户的前一个 HTTP 请求发到特定后端，下一个应该也发到该后端。这至少在最初解决了前面讨论的问题。但是如果该后端死机，负载均衡器将把该用户的请求发送给另一个后端。这是没有选择的。新的后端不知道用户已经登录，会要求用户再次登录。因此，这只是一个不完整的解决方案。

❑ **共享状态**：在这种情况下，用户已经登录的事实和用户的档案信息保存在所有后端都能访问的某个位置。对于每个 HTTP 连接，用户状态从这一共享区域读取。利用这一方法，每个 HTTP 请求发往不同的机器就不要紧了。用户不会在每次切换后端时被要求登录。

❑ **混合**：当用户状态从一个后端转移到另一个后端时，通常会给 Web 服务器造成额外的工作负载。有时候，这种负担很小，可以容忍。但是对于某些应用程序，这一工作很不方便，需要更多处理，因此同时使用粘性连接和共享状态是最佳解决方案。

保存和读取共享状态有许多种方案。一种简单的方法是使用数据库服务器上所有后端都有权访问的一个表。遗憾的是，数据库可能响应缓慢，因为它们没有为此类操作进行优化。其他的一些系统是为共享状态存储专门设计的，它们往往将所有信息存储在 RAM，提供最快速的访问。Memcached 和 Redis 就是两个例子。在任何情况下，都应该避免使用 NFS 服务器上的目录，因为它们在服务器停机时不能提供故障切换，也不提供可靠的文件加锁机制。

4.2.4 用户身份标识

虽然与网站的交互对用户来说似乎是无缝的会话，但是实际上是由许多不同的 HTTP 请求组成的。后端必须知道这么多的 HTTP 请求来自于同一个用户。它们无法使用 HTTP 请求的源 IP 地址：因为网络地址转换（Network Address Translation, NAT）的使用，许多不同的机器往往看起来使用相同的 IP 地址。即使情况不是如此，特定机器的 IP 地址也常常变化：当笔记本电脑从一个 WiFi 网络移动到另一个网络，当移动设备从 WiFi 移到蜂窝网络（或者相反）或者机器关机重启后进入不同网络（有时甚至相同网络的地址也不同）。甚至对于在同一台机器上运行两个 Web 浏览器的用户来说，使用 IP 地址作为身份标识也无效。

作为替代，当用户登录到 Web 应用时，Web 应用生成一个密钥并加入响应中。密钥是随机生成的，只提供给这个 Web 浏览器上的这个用户。未来每当 Web 浏览器向相同的 Web 应用发送 HTTP 请求，也会同时发送密钥。因为密钥不发送给任何其他用户，且难以猜测，Web 应用可以确信这是同一个用户。这种方案被称作 Cookie，密钥往往被称作会话 ID（session ID）。

4.2.5 伸缩性

三层 Web 服务相对单一 Web 服务器有许多优势。它的可扩展性更好，可以添加副本。如果每个副本每秒能够处理 500 个查询，可以持续添加它们直到满足需要的总容量。通过将数据库服务分离到独自的平台，可以独立地扩展。

而且这种模式非常灵活，有许多变种：

- ❑ **副本组：**负载均衡器可以为许多组（不止一组）副本提供服务。每个组可以服务于不同网站，每个组的副本数量可以根据网站要求独立增长。
- ❑ **双负载均衡器：**可以有多个负载均衡器，互为副本。如果一个负载均衡器失效，其他可以接管，这个主题在 6.6.3 节中进一步讨论。
- ❑ **多数据存储：**可以有許多不同的数据存储。每个副本组可以有自已的数据存储，或者所有数据存储都是共享的。数据存储可以各自使用相同或者不同的技术。例如，可以有一个数据存储专用于共享状态，另一个数据存储保存产品目录或者服务所需的其他信息。

4.3 四层 Web 服务

4 层 Web 服务用于有许多具备共同的前端基础设施的单独应用程序（图 4.3）的情况。在这种模式中，Web 请求和平常一样进入负载均衡器，在不同前端中划分流量。前端处理和用户的交互，并将内容传达给应用服务器。应用服务器访问最后一层中的共享数据源。

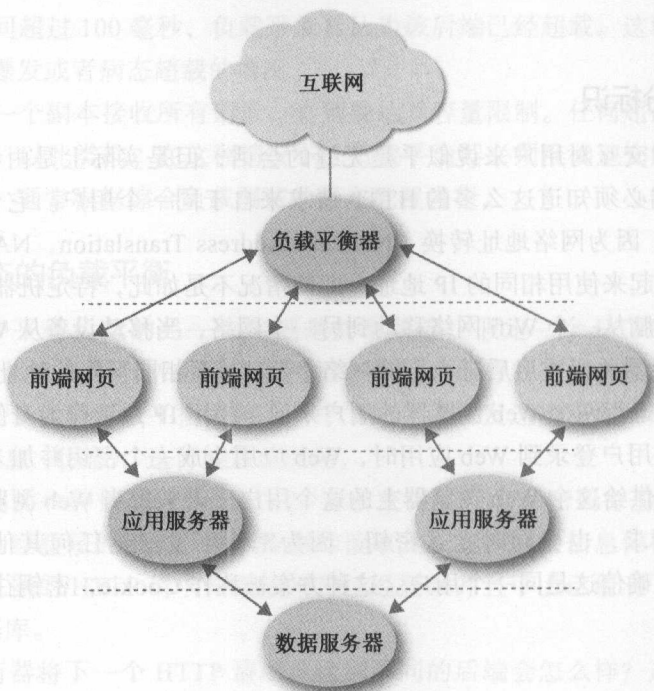


图 4.3 4 层 Web 服务架构

3 层和 4 层设计之间的不同在于应用程序和 Web 服务器在不同机器上运行。使用后一种设计模式的好处是可以消除面向客户交互、协议、安全问题与应用程序的耦合。缺点是应用程序服务团队需要某种信任，才能依赖集中的前端平台团队，而且它还采用了不允许例外的管理规范。

4.3.1 前端

前端负责所有应用程序的共同任务，从而降低了应用程序的复杂度。因为用户交互和应用程序服务解耦，两者可以专注于将一件事情做好。

前端处理所有 Cookie、会话管线（处理同一个 TCP 连接上的多个 HTTP 请求）、压缩和加密。即使应用程序服务器仍然采用 HTTP 1.1，前端也可以采用 HTTP 2.0。实际上，应用程序服务器往往实施 HTTP 协议的较简单子集，因为它们处于完整实现 HTTP 的前端之后。最终用户仍然可以从这些功能中获益，因为前端作为一个网关支持这些功能。

前端软件可以跟踪不断变化的 HTTP 协议定义，而应用程序服务器则不必这么做。HTTP 最初是个简单的协议，处理简单的请求，随着时间的推移，它已经得到发展，拥有比原来丰富得多的功能集，实现起来更复杂、更困难。前端可以独立升级，意味着没有必要等待每个应用服务器升级。

前端处理所有与用户登录和注销相关的事项。在前端层处理这些任务使得所有应用程序很容易拥有统一的用户名和密码基础设施。从前端发往应用层的请求包括预先经过身份验证的用户名。应用程序可以暗中地信任这些信息。

前端往往能够解决问题，使应用程序无须专注这些方面。例如，HTTP 首标是大小写不敏感的，但是有些应用服务器假定首标是大小写敏感的采用不完整实现。前端可以自动将所有首标转换为小写，这样应用服务器只看到它们预期的清晰的协议首标。如果应用服务器不支持 IPv6，前端可以通过 IPv6 接收请求，但是用 IPv4 与后端通信。

有些应用服务器甚至不使用 HTTP。有些公司发明了自己的协议，在前端和应用程序服务器之间更快、更高效地通信。

加密和证书管理

加密集中在一个层次特别重要，因为证书管理相当复杂。非专业人士很容易犯下错误，这些错误至少会弱化系统的安全性，在最坏的情况下，会使服务停止。通过集中加密功能，可以确保管理者是所在领域的专家。每个应用服务器通常由不同团队运营，寄希望于每个团队在加密证书管理以及应用程序上都有高超的能力，是不合理的。

即使所有人都有高度的安全专业能力，仍然有另一个问题：信任。每个管理加密证书的人都必须得到信任，不会暴露或者窃取密钥。信任一个专业团队，比信任许多单独团队的成员们更安全。在某些情况下，所有服务使用相同的密钥，这意味着任何一个团队不小心泄露密钥，都可能弱化所有应用程序的安全性。正如 Cheswick、Bellovin 和 Rubin (2003) 建议的，最佳安全性策略往往是将所有鸡蛋放在一个篮子里，然后确保篮子确实坚固。

安全性上的好处

前端是系统直接暴露到互联网的一部分。这减少了必须防御攻击的场所。在安全领域，这种方法被称作减少“攻击面区域 (attack surface area)”。通过将这些功能与应用程序解耦，可以更快地修复缺陷和安全漏洞。

HTTP 是一个复杂的协议，而且每次修订都变得更加复杂。越复杂的东西，就越可能包含缺陷或者安全漏洞，快速升级前端并独立于任何应用服务器的升级计划是很重要的。应用团队有自己的优先级，可能不愿意或者无法在第一时间内进行软件升级。单独应用和应用团队往往有成百上千个，跟踪所有升级是不可能的。

4.3.2 应用服务器

前端向应用服务器发送查询。因为所有 HTTP 处理由前端负责，可以在前端 - 应用之

间采用 HTTP 之外的协议。HTTP 是一种通用协议，所以它速度缓慢，在服务 API 请求时不像专门构建的协议那么出色。

分离应用服务器和前端还意味着，不同应用程序可以运行在不同服务器上。为前端和每个应用使用专用服务器意味着，每个组件可以单独伸缩。而且这样会带来更高的可靠性，因为一个应用中的问题不会影响其他应用或者前端。

4.3.3 配置选项

将功能分为前端、应用服务器和数据服务层还使人们可以选择适用于每一层次的硬件。前端通常需要高网络吞吐率，而对存储的需求很少。它们还应该布设在网络中的特殊位置，以获得直接的互联网访问。应用服务器可以根据每个应用程序的需求，安装在不同配置的机器上。数据库服务很可能需要大量磁盘存储，或者是一棵服务器树（如 1.3.3 节中所描述的）。

4.4 反向代理服务

反向代理使 Web 服务器可以透明地从另一个 Web 服务器上提供内容。用户看到的的是一个紧密结合的网站，尽管实际上它是由许多应用程序拼凑起来的。

例如，假定某个网站为用户提供体育新闻、天气预报、金融新闻和电子邮件服务，以及一个主页：

- ☐ www.company.com/ (主页)
- ☐ www.company.com/sports
- ☐ www.company.com/weather
- ☐ www.company.com/finance
- ☐ www.company.com/email

上述 Web 功能由差异很大的 Web 服务提供，但是它们可以通过反向代理无缝地组合为统一的用户体验。请求进入反向代理，该代理解释 URL 并从相关的服务器或者服务收集所需的页面。然后，将结果传递给原始请求者。

和前端 4 层 Web 服务一样，这种方案能够实现集中化的安全性和其他服务。在一个域名背后的许多应用程序简化了许多管理性任务，如每个域加密证书的维护。

反向代理和 4 层 Web 服务的前端之间的差别在于，反向代理更简单，通常只是连接 Web 浏览器和在代理之后的 HTTP 服务器组合。有时候，反向代理是独立软件，但是因为它的功能很简单，所以往往是通用 Web 服务器软件的一个特性。

4.5 云规模服务

云规模服务是分布到全球的。服务基础设施使用前面讨论过的某一种架构（如 4 层

Web 服务), 然后在世界各地复制, 使用全球负载均衡器将流量导向最近的位置 (图 4.4)。

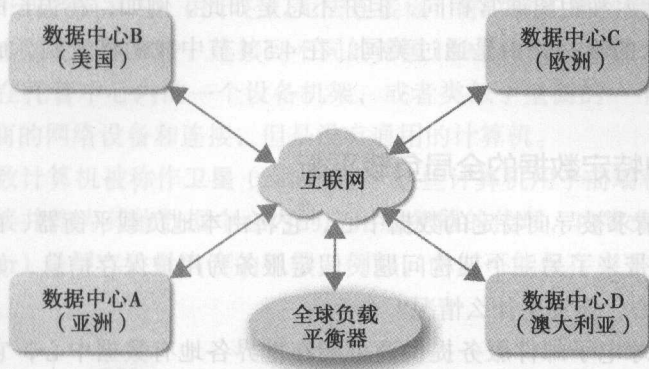


图 4.4 云规模 Web 服务架构

数据包在整个互联网上穿梭需要时间, 距离越远, 时间越长。所有在澳大利亚花费重要时间访问位于美国的电脑的人都会告诉你, 尽管数据以光速穿行, 仍然不够快。这么远距离造成的延迟会使交互性应用程序性能低下, 无法使用。

这一问题的解决方案是使数据和计算更靠近用户。我们在全球构建多个数据中心, 或者在其他人的数据中心内租赁空间, 并在每个数据中心复制服务。

4.5.1 全局负载均衡器

全球负载均衡器 (Global Load Balancer, GLB) 是一个 DNS 服务器, 将流量导向最近的数据中心。DNS 服务器通常对查询返回相同的答案, 而不管查询源自何地。GLB 检查查询的源 IP 地址, 根据源 IP 地址的定位返回不同结果。

定位 (Geolocation) 是确定互联网上的某台机器物理位置的过程, 这是一项很难的任务。电话号码中的国别代码和区号能够提供用户所在位置相当准确的指示 (但是在移动电话的时代就不那么精确了), 与此不同, IP 地址没有具体的地理含义。有一个由某些公司组成的小规模行业, 使用各种手段 (以及许多猜测) 确定每个 IP 子网的物理位置。它们销售用于定位的此类信息的数据库。

4.5.2 全局负载均衡方法

GLB 维护副本、位置及 IP 地址的列表。当 GLB 被要求将某个域名转换为 IP 地址时, 它在确定以哪个 IP 地址作为应答时考虑请求者的定位。

GLBs 使用许多不同的技术:

- 最近: 严格选择和请求者最近的数据中心。
- 有限制的最近: 选择最近的数据中心, 直到该站点满。在那个时候, 选择下一个最近的数据中心。如前所述, GLB 包含了慢启动技术, 原因同本地负载均衡器。

□ **根据其他指标的最近**：最佳位置可能不由距离决定，而是使用其他指标，如延迟或者成本。延迟和距离通常相同，但并不总是如此。例如，在很长的一段时间，大部分南美国家的唯一路由是通过美国。在 4.5.4 节中我们将看到，成本也不总是距离的函数。

4.5.3 使用用户特定数据的全局负载均衡

现在，HTTP 请求被导向特定的数据中心，它将由本地负载均衡器、前端和组成服务的其他设备处理。这带来了另一个架构问题。假定服务为用户保存信息，如果用户的数据被存储在其他数据中心，会发生什么情况？

例如，某个全球电子邮件服务提供商可能在世界各地有数据中心，它可能在用户创建账户时将其电子邮件保存在最靠近的数据中心。但是如果用户迁移了怎么办？或者如果数据中心退役，用户账户被移到其他数据中心，该怎么办？

全球负载均衡器工作于 DNS 级，对用户是谁毫无认知，它无法确定 Mary 发送了 DNS 查询，并返回包含其数据的服务副本的 IP 地址。

这个问题有几种解决方案。首先，通常每个用户的电子邮件都保存在两个不同的数据中心。这样，如果一个数据中心下线，数据仍然可用。现在 Mary 被导向包含其电子邮件的数据中心的概率增大了一倍，但是 HTTP 请求仍然有可能到达错误的数据中心。

为了摆脱这种困境，前端与电子邮件服务通信找出 Mary 的电子邮件所在位置，然后访问正确数据中心内的应用服务器。Web 前端是通用的，但是它们从特定的数据中心获取电子邮件。

为此，公司必须在每个数据中心之间具备连接，使前端可以与任何应用服务器通信。它们可以通过互联网与数据中心通信，但是在这种情况下，公司通常在数据中心之间有私有专用广域网（Wide Area Network, WAN）连接。专用 WAN 为公司提供更多的控制和更可靠的性能。

4.5.4 内部主干网

连接数据中心的私有 WAN 连接组成了**内部主干网（internal backbone）**。内部主干网一般不可见于互联网，是一个私有网络，从许多位置连接到互联网。在有数据中心的地方，数据中心通常连接到该地区的许多个 ISP。直接连接到 ISP 在速度和成本上具备优势。如果到特定 ISP 没有直接连接，向该 ISP 的用户发送数据就必须通过连接你的 ISP 和这些用户的 ISP 的另一个 ISP。这个中间 ISP 被称作**过境 ISP（transit ISP）**。过境 ISP 向其他 ISP 收费，为它们提供允许数据包穿越其网络的特权。在你和客户之间往往有多个过境 ISP，过境 ISP 越多，连接的速度更慢、更不可靠也更昂贵。

POP

接入点（point of presence, POP）是用于连接到本地 ISP 的小规模远程设施。连接到

许多 ISP 很有好处，但是它们不总是能够连接到你的数据中心。例如，你的数据中心可能不在它们运营的州或者国家内。由于它们无法连接你，你必须扩展网络到靠近它们的地方。例如，可以在柏林创建一个 POP，连接到不同的德国 ISP。

POP 通常是在托管中心内的一个设备机架，或者类似于壁橱的一个小空间。它包括来自各个电信提供商的网络设备和连接，但是没有通用的计算机。

POP 加上少数计算机被称作卫星 (satellite)。这些计算机用于前端和内容分发服务。前端终止 HTTP 连接并作为其他数据中心内的应用服务器的代理，内容分发服务器 (Content Distribution Server) 是缓存大量内容的机器。例如，它们可能保存当下最受欢迎的 1000 个视频 (见图 4.5)。

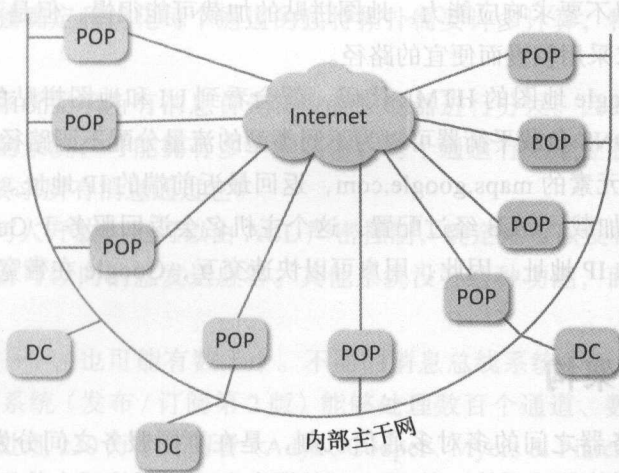


图 4.5 连接许多数据中心和互联网接入点的私有网络主干

因此，内部网络连接数据中心、POP 和卫星。

全球负载平衡与 POP

前面所述的网络带来了一个有趣的问题：GLB 应该将流量导向最近的前端，还是容纳特定服务应用服务器的最近数据中心？

例如，想象（下面的数字都是虚构的）Google 有 20 个 POP、卫星和数据中心，但是 Google 地图仅从 5 个数据中心提供服务。有两条路径可以到达提供 Google 地图服务的服务器。

第一条路径是前往最近的前端。不管前端是在数据中心还是卫星都没有关系。连接的前端很有可能不是托管 Google 地图的 5 个数据中心之一，所以该前端将通过私有主干与实际托管地图服务的最近数据中心通信。这种解决方案非常快，因为 Google 完全可以控制私有主干，该主干设计为提供精确的延迟和必需的带宽。主干网上没有任何其他客户能够占用带宽或者使该连接超载。但是，在主干网上发送的每个数据包都有相关的成本，Google 负责这项开支。

第二条路径是转向托管 Google 地图的最近数据中心内的一个前端。如果该数据中心非常远，查询可能穿越多个 ISP 到达那里，可能还要越过大洋。这种解决方案速度相当慢，越洋线路可能超载，ISP 没有任何动力为其他人的流量提供高性能。尽管如此，传输成本是 ISP 的负担，而不是 Google 的，这种连接的速度可能很慢，但是成本由别人承担。

Google 地图采用哪条路径？快速 / 昂贵的还是经济 / 慢速的连接？回答是：两者都有！

Google 希望为用户提供快速、灵敏的服务。因此，与用户界面（UI）相关的流量通过第一条路径发送。这些数据是 HTML，通常很小。

Google 地图的另一部分是交付地图“拼贴”——组成地图的图形。尽管经过压缩，它们的体积仍然很大，会耗费许多带宽。但是，它们是由非常精巧的 JavaScript 代码从“屏幕之外”加载的，所以不要求响应能力。地图拼贴的加载可能很慢，但是不会影响用户体验。因此，地图拼贴请求采用较慢而便宜的路径。

如果你观察 Google 地图的 HTML 代码，就会看到 UI 和地图拼贴的 URL 引用不同主机名。这样，全球 DNS 负载均衡器可以为不同类型的流量分配不同路径。GLB 的配置是用于与 UI 相关的所有元素的 `maps.google.com`，返回最近前端的 IP 地址。地图拼贴使用包含不同主机名的 URL 加载。GLB 经过配置，这个主机名会返回服务于 Google 地图的最近数据中心内某个前端的 IP 地址。因此，用户可以快速交互，Google 在带宽上的付费也较少。

4.6 消息总线架构

消息总线是服务器之间的多对多通信机制，是在不同服务之间分发信息的方便手段。消息总线正在成为系统管理系统、基于 Web 的服务和企业系统后台使用的流行架构。这种方法比重复轮询数据库以查看是否有新信息到达更高效。

消息总线是由服务器向“通道”（类似于收音机的频道）发送消息，其他服务器监听所需通道的机制。发送消息的服务器是**发布者**（Publisher），接受者为**订阅者**（Subscriber）。服务器可以是某个通道的发布者或者订阅者，也可以简单地忽略该通道。这就可以实现一对多、多对多和多对一通信。一对多通信使一个服务器能够快速地向大量机器发送信息。多对多通信类似于聊天室应用，许多人都可以听到其他人说的话。多对一通信可以实现漏斗形配置，许多机器生成信息，供一台机器使用。一个中央机构（或称控制器）管理哪些服务器连接到哪些通道。

发送的消息可以包含任何类型的数据，它们可以是实时更新的聊天室消息、数据库更新或者更新用户显示以指示收件箱中有消息到达的通知，也可能是传达状态更改、服务请求或者日志信息的低优先级或者批量更新。

消息总线技术有许多名称，包括消息队列、队列服务或者发布 / 订阅（pubsub）服务。例如，Amazon 提供简单队列服务（Simple Queue Service, SQS）、MCollective 被描述为发布订阅中间件，RabbitMQ 自称为消息代理。

消息总线系统的高效之处在于客户端仅在订阅时才会接收消息。它所涉及的机器可能成百上千,但是这些机器的不同子集通常订阅不同的通道,消息仅发往订阅的机器,从而保留了网络带宽和处理能力。这种方法比向所有机器发送所有消息,由接收机器过滤不感兴趣消息的广播系统更高效。消息总线系统对运营部门很友好,连接一个命令行客户端监听消息、查看发送的消息十分简单。这种功能在调试需要探查信息流时很方便。

4.6.1 消息总线设计

消息总线控制器学习底层物理网络拓扑,为每个通道确定消息需要的最短路径。IP 多播往往用于同时向同一子网的许多机器发送消息。IP 单播用于在子网之间或者 IP 多播不可用时传输消息。单独确定和优化每个通道的独特拓扑需要许多计算,特别是在有数千个通道时。

有些消息总线系统要求所有消息首先转发到控制器进行分发。因此,控制器可能成为瓶颈。其他更高级的系统,可能拥有多个控制器,每个通道有一个控制器,或者由控制器控制拓扑,但是不要求所有消息通过它。

通道可以向任何人开放,也可以由 ACL 严密控制,确定谁可以发布、谁可以订阅。在某些系统上,监听者可以向消息发送应答。其他系统没有这种功能,而是创建第二个通道用于发布应答。

通道可能只有一个,也可能有数千个。不同的消息总线系统为不同的规模进行优化。Google 的 PubSub2 系统(发布/订阅第 2 版)能够处理数百个通道、数万台主机。Google 的 Thialfi 系统可以处理 230 万个订阅者(Adya、Cooper、Myers & Piatek, 2011)。

4.6.2 消息总线可靠性

消息总线系统通常保证每条消息都能够被接收,例如,有些消息总线系统要求订阅者答复每条接收到的消息。如果订阅者崩溃,当它返回服务时保证会再次接收到所有未答复的消息。如果答复没有回到消息总线系统,订阅者可能接收到消息的第二个拷贝,检测和跳过重复消息由订阅者决定。

不过,消息总线系统对订阅者长时间停机的处理不同。在某些系统中,在订阅者停机时将会错过消息。在其他系统中,消息会保存某一时长,只在订阅者停机时间超过定义时长才会丢失。有些消息总线系统只在 RAM 中保存消息几分钟,之后就放弃保存并将消息排队写入磁盘。当订阅者上线时,将收到大批积压的消息。

消息总线系统不保证消息将以和发送相同的顺序接收。如果保证这一点,就意味着如果一条消息重试,所有其他消息就必须等待直到它成功。同时,可能有几百万条消息被延迟。

因此,订阅者必须具备处理不按顺序到达的消息的能力。消息通常包含时间戳,这能够帮助订阅者检测何时消息不按顺序到达。但是重新排序消息即便不是不可能,至少也是

很困难的。你可以保存消息直到后续的消息到达，但是如何知道需要等待多长时间？如果等待一个小时，某条缓慢的消息可能在 61 分钟之后到达。因此，最好是编写不依赖完全排序消息的代码，而不是试图重新排序消息。

因为消息可能遗漏或者丢失，服务通常采用不基于消息总线的机制，而是让客户端补上遗漏的内容。例如，如果消息总线用于保持数据库同步，可能有一种途径接收所有数据库键值列表以及最后更新的日期。每天接收一次该列表，订阅者就可以注意到任何遗漏的数据并发出请求。

系统越大，消息总线架构就越具吸引力，因为它们快速、高效，且将控制和运营的责任交给监听者。使用消息总线架构的出色资源之一是 Hohpe 和 Woolf（2003）所著的《Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions》。

4.6.3 例 1：链接缩短网站

链接缩短网站（与 bit.ly 非常类似）的组件使用一个消息总线架构进行通信。该应用有两个用户可见的组件：控制面板（注册需要缩短的新 URL 的 Web UI）和获得短链接并以展开 URL 重定向代码响应的服务。

这家公司希望用户界面和重定向服务器之间能够快速更新。用户通过控制面板创建一个短链接之后立刻尝试使用该链接，以确定其有效的情况很常见。最初，链接创建和重定向服务能够重定向该链接之间有几分钟的延迟。新链接必须索引和处理，然后添加到数据库，数据库的更改必须传播到所有重定向服务器。

为了解决这个问题，该公司建立一个连接所有机器的消息总线系统，该系统有两个通道：一个称作“new shortlinks”（新的短链接），另一个称为“shortlinks used”（用过的短链接）。

如图 4.6 所示，该架构有 4 个元素：

- ❑ 控制面板 Web 服务器：一个 Web 前端，是人们用于创建新的短链接的门户。
- ❑ 主数据库：保存所有短链接信息的数据库服务器。
- ❑ 趋势服务器：跟踪“热门链接”统计的服务器。
- ❑ 链接重定向 Web 服务器：接受短 URL 请求并以展开 URL 的重定向作为响应的 Web 服务器。

任何控制面板服务器从用户那里接收新的短链接时，将把它发布到“new shortlink”（新短链接）通道上。该通道的订阅者包括数据库（存储新的短链接信息）和所有重定向服务器（在其查找的缓存中保存新信息）。执行重定向时，每个重定向服务器依赖来自本地缓存的信息，只在缓存丢失时查询数据库。由于主服务器吸收新的短链接可能花费几分钟或者几小时（因为索引和其他问题），消息总线架构可以使重定向服务器始终保持最新。

当重定向服务器使用短链接时，将把这一事实发布到“shortlinks used”（使用过的短链接）通道。这一通道只有两个订阅者：主数据库和趋势服务器。主数据库更新该数据库条目

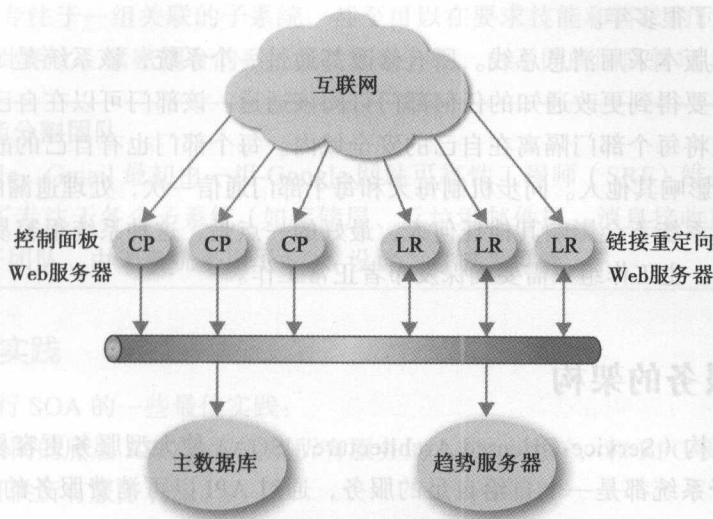


图 4.6 链接缩短架构

的使用统计。趋势服务器维持一个 URL 最近曾经使用过的内存数据库，使该公司始终显示准确的热门 URL 列表。

想象一下，如果没有消息总线，创建这一功能有多么困难。控制面板机器将需要所有重定向服务器的“始终最新”列表，必须处理故障机器、新机器等。当通信没有为网络拓扑进行优化时，实现和消息总线相同的服务质量非常困难。

如果没有消息总线系统，添加新服务意味着更换所有数据提供者，而且还要向其发送一份拷贝。假定设计了一种全新的热门链接计算机制，如果没有消息总线，重定向服务器必须更新以向它们发送信息。而有了消息总线系统，可以添加新的趋势服务器并行运行，而不需要更改其他服务器。如果新服务器证明是成功的，就可以简单地断开旧的趋势服务器。

4.6.4 例 2：员工人力资源数据更新

企业有巨大的员工基础，每天有许多新员工入职、离职、更名或者更改其他数据库属性。同时，这些信息有很多消费者：登录/身份验证系统、工资管理系统、门禁系统和许多其他系统。

在第一代系统中，维护单一数据库，所有系统查询数据库获得更改后的信息。但是随着公司的成长，数据库超载越来越严重。

在第二代系统中，任何需要得到用户更改请求通知的部门必须编写插件，在每次更改时调用。这些插件必须处理添加、更新、删除、更名等子命令。插件都运行于同一个角色的账户下，意味着该账户有权访问公司中的每个关键系统，这是一个噩梦。如果某个部门的插件有缺陷，可能导致整个系统停机。尽管这种系统很脆弱、需要大量工作才能保证运

行，但仍然运营了很多年。

系统的最新版本采用消息总线。所有修改都通过一个系统，该系统是“用户更新”通道的发布者。需要得到更改通知的任何部门订阅该通道。该部门可以在自己的角色账户上运行其系统，这将每个部门隔离在自己的安全域内。每个部门也有自己的故障域，一个监听者的故障不会影响到其他人。同步机制每天和每个部门通信一次，处理遗漏的更新。

新部门加入系统不会影响其他任何人，最好的一点是，这种系统很容易维护，责任被分布到各个部门。主工作组只需要确保发布者正常工作。

4.7 面向服务的架构

面向服务架构（Service-Oriented Architecture, SOA）使大型服务更容易管理。利用这种架构，每个子系统都是一个自给自足的服务，通过 API 以可消费服务的形式提供功能。各种服务通过 API 调用相互通信。

SOA 的目标之一是使服务松散耦合。也就是说，每个 API 以高度的抽象表示其服务。这样，改善甚至替换服务都更加容易。替换必须提供相同的抽象，松散耦合的系统不知道作为架构的一部分的其他系统的内部工作原理。如果它们知道，相互之间的联系就很紧密。

举个例子，想象一个作业调度服务。它接受请求执行各种操作，调度、协调操作并在执行中报告进度。在紧密耦合的系统中，API 与作业调度程序的内部工作紧密联系。API 的用户可能超出需要，规定与作业工作方式相关的细节。例如，API 可能提供对用于避免工作执行两次的加锁机制状态的直接访问。

假定有人提出一种避免重复作业执行的新型内部设计，采用其他加锁方式。如果不更换使用 API 的所有服务代码，就无法完成这一更改。在松散耦合系统中，API 将在更高级抽象上提供作业状态：作业等待、作业运行、作业在哪里运行、能否停止等等。不管内部实现如何，都能够处理这些请求。

4.7.1 灵活性

SOA 使服务的互操作更加简单，服务能以许多出人意料的方法组合，以创建新应用。只要 API 符合新应用的需求，设计中就不需要参考所有其他服务。每个子系统可以作为离散系统管理。管理少数小型、容易理解的服务，比管理具有定义不清的内部联系的大系统更简单。当服务之间的接触点是实施高层抽象、定义良好的 API，服务的演化和替换都会更容易。

4.7.2 支持

SOA 在人员管理上也有好处。随着系统成长，开发和运营团队通常也会成长。大团队比小型的针对性团队更难管理。采用 SOA，很容易在团队成长超越可控限制时进行分割。

每个团队可以专注于一组关联的子系统，甚至可以在要求技能和需求的时候在团队之间交换子系统。与此相比，紧密耦合的系统无法轻松分割，由不同的团队管理。

按照职能分割团队

在 Google, Gmail 最初由一组 Google 网站可靠性工程师 (SRE) 维护。随着系统的成长，划分了专注于各个子系统（如存储层、反垃圾邮件层、消息接收系统、消息交付系统等）的子团队。由于系统采用了 SOA 设计，这成为可能。

4.7.3 最佳实践

下面是运行 SOA 的一些最佳实践：

- ❑ 使用相同的底层 RPC 协议实现所有服务的 APIs。这样，和 RPC 机制相关的任何工具都可供所有服务利用。
- ❑ 采用一致的监控机制。所有服务应该以相同方式向监控系统输出计量指标。
- ❑ 尽可能多地在各种服务中使用相同的技术。使用相同的负载平衡系统、管理技术、编码标准等。由于服务在团队之间流动，如果这些细节一致，人们就更容易快速处理。
- ❑ 采用某种形式的 API 治理。需要设计如此之多的 API 时，维护工作方式的标准就变得很重要。这些标准往往从在过去的痛苦失败中学到的知识中得以传授，这些失败是组织所不希望重复的。

当紧密耦合的系统变得难以维护时，选择之一是将其演化为松散耦合的系统。新推出的系统开始时往往是紧密耦合的，理由是资源效率更高或者开发速度更快（可能是事实，也可能不是）。解除组件之间的耦合可能是漫长而艰难的旅程。首先逐个识别可能分割为服务的组件。不要选择最容易的组件，而是最需要 SOA 好处（灵活性、易于升级和替换等）的组件。

4.8 小结

基于 Web 的应用程序必须成长和扩展。小的服务可以在单一机器上运行，提供静态、动态或者数据库驱动的内容。达到一台机器的上限时，可以使用 3 层架构。这种架构将每个功能移到不同的机器上：一个负载平衡器将流量导向 Web 服务器，一个或者多个数据服务器提供静态或者数据库内容。

本地负载平衡器在数据中心内的副本之间分布流量。它们拦截流量并将其重定向到 Web 副本。有许多负载平衡技术，以及许多确定流量分布方式的算法。当应用程序在副本中划分时，用户身份和其他状态信息的同步变得复杂。这一目标通常用某种保存所有副本公用状态的服务器实现。

4 层架构为许多 3 层系统创建一个前端。新层次处理公共服务，这些服务往往与用户会话、安全性和日志记录有关。反向代理确保许多应用服务器看起来像单一的大应用程序。

云规模服务采用这种架构，并将其复制到全世界的许多数据中心。它们使用全球负载均衡器将流量导向特定的数据中心。如果能够从控制国际数据中心的网络质量获益，可以在数据中心之间使用私有网络。私有网络可能从许多接入点连接到互联网，改善许多 ISP 的连接性。接入点（POP）可能有终止 HTTP 会话、用更高效的协议将数据转发给数据中心的服务器。

其他架构模式适合于非 Web 应用。消息总线架构创建消息传递系统，消除通信与需要信息的服务之间的耦合。面向服务架构包含许多小型服务，它们互操作以创建更大的服务。SOA 中的每个服务可以单独伸缩、升级甚至替换。

第 1 章讨论了构成、服务器树和其他模式。这些基本要素组成了本章讨论的许多模式。第 5 章将讨论数据存储模式，特别是 5.5 节将描述分布式散列表。

这些架构模式在成本、伸缩性和故障弹性上都有妥协。理解这些妥协是决定每种模式使用时机的关键。

练习

1. 描述单机、3 层和 4 层 Web 应用架构。
2. 描述使用数据库生成内容的单机 Web 服务器如何演化为 3 层 Web 服务器。怎样做才能造成的停机时间最短？
3. 描述常见的 Web 服务器架构，按照从小到大排序。
4. 描述不同本地负载均衡器类型的工作原理及其优缺点。你可以选择制作一个比较图表。
5. 什么是“共享状态”，如何在副本之间维护？
6. 4 层架构中第 1 层提供什么服务？
7. 反向代理有何作用？何时需要它？
8. 假定你想要构建简单的图像共享网站，如果网站的意图是服务于世界上某个地区的人们，将如何设计？如何扩展它，使其在全球工作？
9. 什么是消息总线架构，如何使用？
10. 什么是 SOA？
11. 为什么 SOA 是松散耦合的？
12. 如何将电子邮件系统设计为 SOA？
13. Christopher Alexander 是谁，他对架构有何贡献？

伸缩性设计模式

伸缩性是真正的问题，其他都是小问题。

——O'Dell 公理

系统伸缩能力是处理日益增长的工作负载的能力，工作负载通常用每秒事务量、总数据量或者用户数计量。系统的伸缩性有一定的限制，达到这一极限之后，需要重新设计以容纳更大的增长。

分布式系统必须从一开始就以可伸缩的方式构建，因为增长是可以预期的。不管构建的是基于 Web 的服务还是批处理数据分析平台，目标始终是成功吸引更多的用户、使用量或者数据。

确保服务快速并保持快速是至关重要的。如果服务没有伸缩性，或者在更流行时变得太缓慢，用户将转投别处。Google 发现，在搜索结果中人为插入 400 ms 的延迟，将使用户少进行 0.2%~0.6% 的搜索，这可能转换为几百万甚至几十亿美元的收入损失。

有讽刺意味的是，缓慢的 Web 服务比故障停机更令人沮丧。如果网站下线，用户立刻就了解了这一事实，可以进入竞争网站或者找其他的事情做，等待它恢复。如果网站缓慢，用户的体验就只剩下痛苦和沮丧。

可伸缩系统的构建不是在无意中达成的。分布式系统不会自动具备可伸缩性，初始的设计就必须以伸缩性为目标，满足服务的需求，而且还要包含为未来的成长创建选项的特性。一旦系统进入运营，我们要不断地优化系统，以帮助它实现更好的伸缩性。

在前几章中，我们已经讨论了使分布式系统成长为极大规模的许多种技术。本章，我们将更加详细地回顾这些技术，复习与伸缩性相关的术语，研究伸缩性技术背后的理论，并描述伸缩所用的具体技术。描述系统性能和规模的数学术语可以在附录 C 中找到。

5.1 总体战略

构建可伸缩系统的基本战略是从设计的开始就牢记可伸缩性，避免可能妨碍未来扩展的设计元素。

初始需求应该包含预期规模的近似描述：存储数据的大小、数据处理系统的吞吐量、服务当前接收的流量、预期的增长率。所有这些因素对设计都有指导意义，这一过程在 1.7 节中曾经描述过。

一旦系统投入运行，就会发现性能的极限。这时就需要实现进一步伸缩性的设计特性。虽然为预见潜在伸缩性问题做了各种努力，但是并不是所有问题都能够引起设计者的注意。处理未来潜在伸缩性问题的附加设计及编码工作在优先级上低于修复眼下直接问题的代码编写。花费太多时间避免不一定会发生的伸缩性问题，被称作过早优化（Premature Optimization），应该避免。

5.1.1 识别瓶颈

瓶颈是系统中发生拥塞的地方，是资源耗尽而对性能造成限制的位置。每个系统都有瓶颈，如果系统表现不佳，修复瓶颈可以使系统性能更好。即使系统表现良好，了解瓶颈的位置也是有用的，因为这使我们能够预测和避免未来的问题。在这种情况下，生成附加负载（可能在测试环境中）观察哪部分性能受到影响，就可以找到瓶颈。

确定可伸缩性就是找出系统中的瓶颈并将其消除。瓶颈是积压工作形成的位置。对瓶颈的上游处理进行优化，只会使积压工作更快增长。对瓶颈下游进行优化可能改善那部分的效能，但是不能改善系统的总吞吐率。因此，不以瓶颈为焦点的任何努力都是徒劳的。

5.1.2 重新设计组件

有些伸缩性问题可以通过调整当前系统解决。例如，增大缓存可能和调整配置文件一样简单。其他伸缩性问题则需要设计工作。

重写系统的各个部件称作再工程（Reengineering），通常是为了改善速度、功能性或者资源消耗。有时候，再工程是很困难的，因为之前的设计决策导致了特定的代码或者设计结构，最好的做法往往是首先用功能等价但内部组织便于其他再工程工作的代码代替这些代码。重新构造现有的代码主体——修改内部结构，而不改变其外部表现——称作重构（Refactoring）。

5.1.3 计量结果

伸缩性解决方案必须用证据（即从真实系统收集的数据）加以评估。计量、尝试解决方案并重复相同的计量以观察效果。如果效果很小甚至为负，就不能部署这一解决方案。

例如，如果性能缓慢且计量指标显示缓存命中率很低，缓存可能太小。在这种情况下，

我们应该计量性能, 改变缓存大小, 然后再次计量性能。虽然经过这样的调整缓存的性能可能更好, 但是对系统整体可能不会带来显著的改善, 或者改善没有大到足以弥补需要的额外 RAM 的成本。

如果在更改前后没有计量, 就无法确定更改是否真正产生效果。在更改时不做计量, 系统管理在最好的情况下依靠的是运气, 在最糟的情况下则全凭自负。匆忙尝试一种解决方案, 在更改之后才做计量往往很有诱惑力, 但是这和完全不做计量一样有害, 因为完全没有比较的基准。

过去的经验应该能够提供指导, 但是我们必须抵制诱惑, 绝不忽略用计量和分析指导决策的科学过程。分布式系统总是很大, 任何个人都无法“尽知”正确的做法。经验丰富的系统管理员的直觉和猜测也不如初级人员花费时间进行科学分析后做出的建议: 建立数据收集机制、计量、验证关于错误所在的假设、测试可能修复问题的理论并分析结果。

5.1.4 保持主动

修复瓶颈的最佳时机是在它成为问题之前。理想情况下, 修复应该在这一时刻即将到来之前及时进行。如果我们过早修复问题, 可能将精力浪费在从未发生的问题上——将精力放在其他地方可能更好。如果我们太晚开始设计和实施解决方案, 问题就会在解决方案部署之前发生。如果等待的时间太长, 问题就会在我们毫无防备的情况下出其不意地发生, 我们只能以“救火队”的方式寻找解决方案。设计需要时间, 在匆忙之中进行设计会导致错误和更多问题。我们想要的是适中的解决方案: 不早不晚, 刚刚好。

每个系统都有瓶颈或者约束条件, 这不是坏事。约束条件是所有系统固有的, 它们规定了处理的速度或者处理中的工作量。如果当前速度足够, 瓶颈就不成为问题。换言之, 约束条件只在真正束缚系统实现目标的能力时候才成为问题。

实现运行系统伸缩性的最佳策略是提前足够久的时间预测问题, 以便留出足够的时间设计合适的解决方案。这意味着, 应该始终收集足够的计量指标, 了解瓶颈所在。这些指标应该经过分析, 以便预测何时这些瓶颈会成为问题。例如, 简单地计量消耗的互联网带宽并制作图表, 便于预测何时互联网连接的容量会被耗尽。如果订购更多带宽需要 6 周的时间, 就必须至少提前 6 周下订单, 最好提前 12 周, 使失败的尝试有时间返工。

有些解决方案可以通过调整配置的设置快速实现, 其他方案则需要花费几周或者几个月的设计工作才能解决, 往往涉及重新编写或者替换主要系统。

5.2 纵向扩展

扩展系统的最简单方法是使用更大、更快的设备。运行太慢的系统可以转移到具备更快的 CPU、更多的 CPU、更多的 RAM、更快的磁盘、更快的网卡等设备的机器上。现有计算机往往可以在上述某个属性上进行改善, 而无须替换整个系统。这被称作纵向扩展

(scaling up), 因为系统在规模上增大。当这种解决方案很有效时, 往往是最简单的解决方案, 因为不需要重新设计软件。但是, 这种扩展方式有许多问题。

首先, 系统的规模是有限制的。所能获得的最快、最大、最强计算机可能不足以承担手上的任务。没有任何计算机可以存储 Web 搜索引擎的整个语料库, 或者具备能够处理 PB 级数据集、响应每秒数百万个 HTTP 查询的 CPU 能力。当今市场上可以获得的机器总有极限。

其次, 这种方法并不经济。速度提高一倍的机器, 其成本不止两倍。这样的机器的销售量不大, 因此生产的规模也不大。购买最新的 CPU、磁盘驱动器和其他组件需要额外的花费。

最后 (也是最重要的), 纵向扩展并不适合于所有情况。如果不改变使用的软件设计, 购买更快、更强的机器往往不能得到按比例放大的吞吐率。单线程软件在具备多处理器的机器上不会更快地运行。当 CPU 数量超出某一数量时, 由于锁争用等瓶颈, 为在所有处理器上分布任务编写的软件可能没有太多的性能改进。

同样, 改善任何一个组件的性能, 不能保证改善整个系统的性能。例如, 如果协议在延迟很高时性能低下, 更快的网络连接不能改善吞吐率。

附录 B 更详细地介绍这些问题以及这些问题导致分布式计算发明的历史。

5.3 AKF 伸缩立方体

大比例扩展的方法归结为 3 个基本选项: 复制整个系统 (水平复制); 将系统分割为单独的功能、服务或者资源 (功能或者服务分割); 将系统分割为单独的块 (查找或者公式分割)。

图 5.1 中的 AKF 伸缩立方体是由 Abbott、Keeven 和 Fisher 开发的, 将上述类别概念化为 x、y 和 z 轴 (Abbott & Fisher 2009)。

5.3.1 x 轴: 水平复制

水平复制 (Horizontal Duplication) 通过复制服务增大吞吐率, 也称为水平扩展 (Horizontal Scaling) 或者向外扩展 (Scaling Out)。

这类复制在前面几章已经做了讨论。例如, 在负载均衡器之后使用许多 Web 服务器副本的技术就是水平扩展的一个例子。

一组共享资源称作资源池, 在池中添加资源时, 每个副本必须能够处理相同的事务, 得出相同或者等价的结果。

如果数据增加或者有需要特殊处理的复杂处理时, x 轴无法很好地扩展。如果每个事务可以在所有副本上独立完成, 性能改善可能与副本数量成正比, 效率不会有大规模的损失。

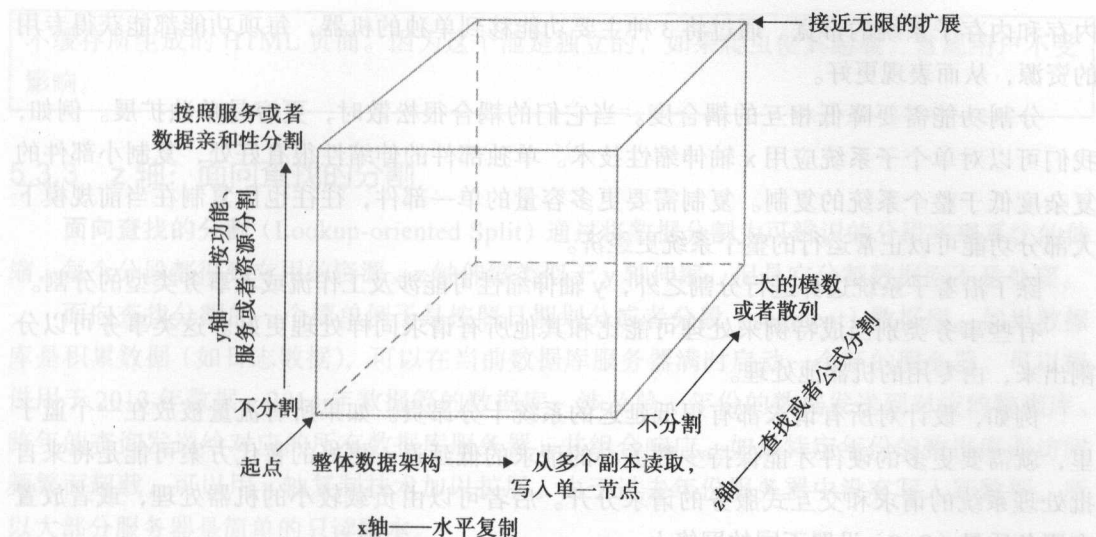


图 5.1 AKF 伸缩立方体。AKF Partners 注册商标，来自《Scalability Rules: 50 Principles for Scaling Web Sites》

关于伸缩性的推荐图书

Abbott 和 Fisher 所著的《The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise》(2009) 包含了大量人、过程和技术伸缩性的技巧与讨论。

Abbott 和 Fisher 的另一本著作《Scalability Rules: 50 Principles for Scaling Web Sites》(2011) 较为精简，聚焦于技术策略和技术。

当事务要求副本通信时，伸缩的效率较低。例如，写入新数据的事务必须传达给所有副本，这可能要求所有副本推迟与更新相关的所有未来事务，直到所有副本都接收到更改。这与 CAP 原则相关（在 1.5 节中讨论过）。

x 轴扩展所涉及的技术包括：

- ☐ 添加更多机器或者副本。
- ☐ 添加更多磁盘主轴。
- ☐ 添加更多网络连接。

5.3.2 y 轴：功能或者服务分割

功能或者服务分割（Functional or Service Split）意味着通过分割每个单独功能，使其能够分配更多资源而扩展系统。

这种伸缩性的例子在 4.1 节中讨论过，我们有一台用于 Web 服务器、数据库和应用服务器（用于动态内容生成）的机器。这 3 项功能都竞争磁盘缓存、CPU 等资源，以及磁盘、

内存和内存子系统的带宽。通过将 3 种主要功能移到单独的机器，每项功能都能获得专用的资源，从而表现更好。

分割功能需要降低相互的耦合度。当它们的耦合很松散时，更容易单独扩展。例如，我们可以对单个子系统应用 x 轴伸缩性技术。单独部件的伸缩性很有好处，复制小部件的复杂度低于整个系统的复制。复制需要更多容量的单一部件，往往也比复制在当前规模下大部分功能可以正常运行的整个系统更经济。

除了沿着子系统边界进行分割之外，y 轴伸缩性可能涉及工作流或者事务类型的分割。

有些事务类别当成特例来处理可能比和其他所有请求同样处理更好。这类事务可以分割出来，由专用的机器池处理。

例如，设计对所有请求都有很低延迟的系统十分昂贵。如果所有流量被放在一个篮子里，就需要更多的硬件才能保持少数关心的请求的低延迟。更好的替代方案可能是将来自批处理系统的请求和交互式服务的请求分开。后者可以由负载较小的机器处理，或者放置在服务质量（QoS）设置不同的网络上。

也可以标记特殊的客户，提供特殊待遇。当用户在金融服务公司投资数百万美元时，该公司的网站设置一个 Cookie。Web 负载平衡器检测该 Cookie，将其流量发送到专门用于重要客户的服务器池中。

另外，可能有一类不常出现的事务特别繁重，将其转移到自己的资源池可以避免一般事务超载。例如，某个特殊查询可能对缓存基础设施造成负面影响。这些查询可以专门由一组使用不同缓存算法的机器处理。

y 轴伸缩性涉及的技术包括：

- ☐ 按照功能分割，每项功能使用单独的机器。
- ☐ 按照功能分析，每项功能使用自己的机器池。
- ☐ 按照事务类型分割。
- ☐ 按照用户类型分割。

案例研究：ServerFault.com 流量类型分割

ServerFault.com 是一个系统管理员的问答网站。向登录用户显示问题（及其答案）时，页面为特定人员扩增和定制。匿名用户（未登录用户）看到的都是相同的通用页面。

为了在 y 轴上扩展系统，将生成相同页面的两种方法分开。匿名页面现在由不同系统处理，生成 HTML 一次并缓存供未来请求使用。由于绝大部分请求来自匿名用户，这种分割大大改善了性能。

匿名页面的查看以另一种方式细分。Google 和必应（Bing）等搜索引擎爬行 Serverfault.com 的每个页面，查找新内容。由于这种爬行会到达每个页面，可能使服务超载，因为请求量和按顺序访问每个网页都会耗尽缓存。这两个因素会导致其他用户遭遇低下的性能。因此，来自 Web 爬虫的请求被发送到专用的副本池。这些副本被配置为

不缓存所生成的 HTML 页面。因为这个池是独立的，如果爬虫使其超载，常规用户不受影响。

5.3.3 z 轴：面向查找的分割

面向查找的分割（Lookup-oriented Split）通过将数据分割为可辨识的分段实现系统的伸缩，每个分段都得到专用的资源。z 轴伸缩类似于 y 轴伸缩，但是它分割数据而不是处理。

面向查找分割的一个简单例子是按照日期划分或者分段（segment）数据库。如果数据库是积累数据（如日志数据），可以在当前数据库服务器满时启动一个新的服务器。可以提供用于 2013 年数据、2014 年数据等的数据库。涉及单一年份的数据发送到对应的数据库。跨年的查询发送给对应的所有数据库服务器，并组合响应。如果特定年份的数据库因访问频繁而超载，可以用 x 轴复制技术加以扩展。由于过去年份服务器中没有写入新数据，所以大部分服务器是简单的只读副本。

案例研究：Twitter 的早期数据库架构

Twitter 初创时，所有推文历史可以放入运行 MySQL 的单个数据库服务器。当服务器满时，Twitter 启动新数据库服务器，修改其软件应对数据按照日期分段的情况。

随着 Twitter 越来越流行，新分段启动和新数据库满之间的时间间隔越来越短。运营团队不得不紧追需求的脚步，这种解决方案无法持续。由于旧机器的流量不多，负载不平衡。这种解决方案也很昂贵，因为每台机器都需要许多副本，后勤支持也很复杂。

最终，Twitter 转向自创的基于 Gizzard 的 T-bird 数据库系统，这种数据库自动平滑伸缩。

分段数据的另一种方法是按地理位置分段。在全球服务中，在世界各地建立许多单独的数据存储是常见的做法。每个用户的数据被保存在最近的存储。这种方法还使用户能够更快地访问到自己的数据，因为数据存储更靠近他们。

从未分段数据库到分段数据库可能需要对应用程序代码进行大量重构。因此，z 轴伸缩往往只在 x 轴和 y 轴伸缩已经到达极限时采用。分段数据的其他方法包括：

- 按照散列前缀：这种方法被称作分片（Sharding），将在后面讨论。
- 按照客户功能：例如，eBay 按照产品（汽车、电器等）分段。
- 按照利用率：将高使用率用户放入专用分段。
- 按照组织部门：例如，销售、工程、业务开发。
- 层次化：分段保持层次结构。DNS 使用这一模式，查找 www.everythingsysadmin.com 之类的地址时首先查找根服务器，然后查找 com 服务器，最后查找 everythingsysadmin 域的服务器。
- 按照随机分组：如果一个机器群集能够可靠地扩展到 50 000 个用户，则为每 50 000

个用户启动新群集。电子邮件服务往往使用这种策略。

5.3.4 组合

许多伸缩性技术组合了 AKF 伸缩立方体的多个轴。下面是一些例子：

- **分段 + 副本**：较频繁访问的分段可以有更大的深度复制。这可以扩展到更大的数据集（更多分段）和更好的性能（每个分段更多副本）。
- **动态副本**：动态添加和删除副本，实现必要的性能。如果延迟太高，添加副本，如果利用率太低，删除副本。
- **架构更改**：根据需求将副本移到较快或者较慢的技术。不经常访问的分片被转移到较慢、较便宜的技术（如磁盘）。要求较高的分片被转移到更快速的技术（如固态硬盘（SSD））。极旧的分段可以归档到磁带或者光盘。

5.4 缓存

缓存是使用快速 / 昂贵媒体的小型数据存储，其意图是改善更大的慢速 / 经济数据存储的性能。例如，最近的数据库查询可能保存在 RAM 中，以便在相同查询重复进行时避免磁盘访问。缓存本身是一种独特的模式，可以视为 AKF 伸缩立方体 z 轴的优化。

考虑超大数据表中的查询，如果该表保存在 RAM 中，查询就会很快。假定数据表大于 RAM 的容量，则只能保存在磁盘上。在磁盘上查询很慢，为了改善性能，我们分配一定数量的 RAM 作为缓存。现在查找的时候，首先检查结果是否可在缓存中找到，如果可以找到则使用该结果，这称作**缓存命中**（cache hit）。如果没有找到，从磁盘进行常规查找，这称作**缓存未命中**（cache miss）。结果照常返回，并保存在缓存中，以加速未来的重复请求。

图 1.10 列出的性能比较有效地评估了缓存命中与缓存未命中的速度。例如，数据库在荷兰，而你身在美国加利福尼亚，如果需要的是少于 10 次的缓存和 2~3 次 1 MB 磁盘的读取，基于磁盘的缓存会更快。相反，如果数据库查询在同一个数据中心进行，必须使用比上述情况快得多的缓存，如 RAM 或者同一子网的缓存服务器。

5.4.1 缓存效能

缓存效能由**缓存命中率**（cache hit ratio rate）计量。该比率是缓存命中数量与总查找数量的比值。例如，如果执行 500 次查找，其中 100 次的结果来自缓存，则缓存命中率为 1/5（20%）。

性能

如果缓存命中节约的时间超过额外开销损失的时间，缓存对性能就有好处。我们可以用加权平均值来估算，如果典型的常规查找时间为 L，缓存命中时查找时间为 H，缓存未命中时查找时间为 M，缓存命中率为 R，则 $H \times R + M \times (1 - R) < L$ 时使用缓存更有效。

如果缓存查找和更新时间极快或者接近于 0，在工程估算时可以简化该公式。实际上，我们可以认为性能上的好处是缓存命中率的函数。例如，假定典型的查找需要 6 s，预测的缓存命中率为 33%。我们知道，在具备即时、零开销缓存的理想世界中，平均查找时间将下降 33%——4 s。这为我们提供了用于规划的最佳情境。实际上，性能的改善会稍微低于这个理想值。

成本效益

缓存带来的好处大于实施缓存的成本时，它才是有成本效益的。RAM 的访问速度快于磁盘，但是成本也高得多。图 1.10 中的性能和成本项不可避免地会随时改变。我们建议根据特定环境中的性能和成本构建自己的图表。

假定没有缓存的系统需要 20 个副本，但是使用缓存只需要 15 个。如果每个副本是一台机器，缓存的成本低于 5 台机器的购置成本，那么它就更有成本效益。

购买价格不总是唯一的考虑因素，如果提供更快服务能够提高 20% 的销售额，则成本应该根据这种改进加权。

5.4.2 缓存布置

缓存可以放在本地，在这种情况下软件执行自己的缓存，在每次缓存命中时节约查询请求问题时间（图 5.2a）。缓存也可以在应用程序外部，部署在服务器和外部资源之间。例如，Web 缓存处于 Web 浏览器和 Web 服务器之间，在可能的情况下拦截请求并缓存（图 5.2b）。缓存还可以处于服务器端。例如，提供 API 用于查找某些信息的服务器可以维护自己的缓存，在必要时为请求提供服务（图 5.2c）。如果有多个缓存，其中一些可能已经过时。这时适用 1.5 节中描述的 CAP 原则。

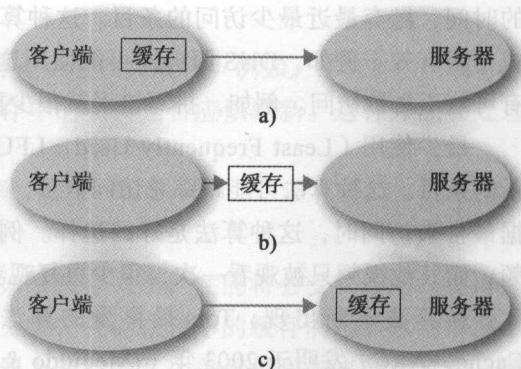


图 5.2 缓存布置

并不是所有缓存都在 RAM 中。缓存介质必须快于主媒体，磁盘可以作为必须从远程服务器收集的数据的缓存，因为磁盘通常快于远程读取。例如，YouTube 视频缓存存在全世界的许多服务器上以节约互联网带宽。极快的 RAM 可以作为常规 RAM 的缓存。例如，CPU 的 L1 缓存是计算机主存储器的缓存。缓存不仅仅用于改善数据查找，例如，计算结果也可以缓存。进行高深的数学计算的函数可以缓存最近的计算结果，这些结果可能会被再次请求。

5.4.3 缓存持久性

当系统启动时，缓存通常为**空或冷却**（cold）。缓存命中率将会很低，在足够的查询使缓存**变暖**（warmed）之前，性能会很低下。有些缓存是持久的，也就是说在重启之后仍然

存在。例如，保存在磁盘上的缓存在重启之后依然存在，RAM 是不持久的，在重启之后会丢失。如果系统的缓存暖化很慢且保存在 RAM 中，在关机之前将其保存到磁盘，启动时读回可能有好处。

案例研究：将 RAM 缓存保存到磁盘

Stack Exchange 网站依赖于一个数据库，该数据库由 Redis 实现 RAM 缓存。如果 Redis 重启，网站性能在 10~15 分钟的缓存暖化期间令人难以接受。Redis 采用了许多特性来避免这一问题，包括将 RAM 中的数据快照保存到磁盘，或者使用第二个 Redis 服务器保存缓存拷贝。

5.4.4 缓存置换算法

处理缓存未命中时，常规查找收集的数据加入缓存，如果缓存已满，必须抛弃某些数据以腾出空间。有许多不同的置换算法用于处理缓存操纵。

一般来说，较好的算法记录更多使用信息，以改善缓存命中率。不同算法适合于不同的数据访问模式。最近最少使用（Least Recently Used, LRU）算法跟踪每个缓存条目使用的时间，抛弃最近最少访问的条目。这种算法在查询经常于较小时间间隔内重复的访问模式中很有效。例如，DNS 服务器可能使用这种算法：如果一个域长时间没有被访问，就很有可能不再被访问。例如，拼写错误就很少重复，最终会在这种缓存中过期。

最少使用（Least Frequently Used, LFU）算法计算缓存条目的访问次数，抛弃最不活跃的条目。这种算法可能跟踪总访问数量，或者保留每小时/每天的计数。当较为流行的数据最常被访问时，这种算法是好的选择。例如，视频服务可能缓存常被观看的某些流行视频，而其他视频只被观看一次，很少重复观看。

新算法不断出现。Tom 最喜爱的算法——自适应置换缓存（Adaptive Replacement Cache, ARC）发明于 2003 年（Megiddo & Modha 2003）。大部分算法对于其他时候很少使用的数据的突然涌入都无能为力。例如，数据库备份逐个读入所有记录，使缓存中充满很少使用的数据。此时，该缓存变冷，性能受到很大影响。ARC 将新缓存数据置于“试用”状态，解决了这个问题。如果缓存第二次被访问，它就离开试用区域进入主缓存。单次遍历数据库会消除试用缓存，但是不会影响主缓存。

还有许多其他算法，但是大部分都是 LRU、LFU 和 ARC 的变种。

5.4.5 缓存条目失效

当主要存储位置中的数据变化时，任何相关的缓存条目就会过时，有许多方法可以处理这种情况。

一种方法是忽略这种情况。如果主存没有变化，缓存条目不会过时。在其他情况下，缓存很小，过时的条目最终会通过缓存置换算法替换。如果系统可以容忍偶然的过时信息，

这种方法就足够了。这种情况很少。

第二种方法是在数据库更改时使整个缓存无效。这种方法的好处是实现很简单。但是这使得缓存冷却，在再次暖化之前性能遭到严重损失。这在缓存快速暖化的应用程序中可以接受。

主存可以和需要无效化的缓存通信，在缓存中的数据变化时使任何条目无效。如果有许多缓存且更新频繁发生，这可能造成许多工作量。这也是1.5节中描述的CAP原理起作用的一个示例：为了达到完美的一致性，涉及条目更新的查询处理必须暂停，直到所有缓存得到通知。

有些方法消除了主存与缓存直接通信的需求，例如，缓存可以假定某个条目只在固定的秒数内有效，可以在每个缓存条目创建时记录一个时间戳，在固定时间之后过时。

另外，服务器可以在应答查询时加入条目可能缓存的时长。例如，假定DNS服务器不仅应答所有查询，还包含应答中每个条目缓存的秒数，这称作存活时间（Time to Live, TTL）值。同样，HTTP服务器可以用到期日作为响应的注释，表示该条目可被缓存的时长。如果不能控制客户端软件，就无法确定客户端遵守这些指令。许多ISP至少缓存DNS条目24小时，这让使用DNS作为全球负载均衡器一部分的人们十分沮丧。许多Web浏览器对忽略HTTP服务器发送的到期日是有责任的。

最后，缓存可以轮询服务器，查看本地缓存是否应该无效化。例如，HTTP客户端可以查询服务器某个项目是否已经更改，以发现缓存中的项目是否应该刷新。这种方法在处理新鲜度查询时明显比完整应答发送更快速。

5.4.6 缓存大小

选择合适的缓存大小很难，但是很重要。如果缓存太小，性能将会受到影响，有可能得到比完全没有缓存更糟糕的结果。如果缓存太大，我们在较小的缓存满足需求的情况下浪费了金钱。如果我们有几千台机器，每台机器上浪费的几个美元累积起来就很可观。缓存通常是固定大小的，这一大小通常由固定的缓存条目或者固定的总存储量组成，在后一种情况下，保存的条目数量可能要相应地进行调整。

选择缓存大小有多种方法，要确定指定情况下的缓存大小，最精确的方法是对运行中的系统进行计量。缓存很复杂，特别是置换算法和缓存大小的相互作用。计量运行系统涉及所有此类因素，但是并不总能做到。

一种方法是运行包括各种缓存大小的系统，计量每一个系统实现的性能。这种基准测试可以使用真实或者模拟数据，在为基准测试准备的单独系统上进行。在真实系统上进行此类操作可以得到最准确的计量指标，但是这样做可能很危险。限制风险的方法之一是只在许多副本中的一个上调整缓存大小，这样负面影响就可能不会被注意到。

另一种方法是为系统配置大于应用程序可能需要的缓存尺寸。如果缓存条目很快过期，缓存将增大到某一尺寸之后停止增长。这可以得出所需缓存大小的上界。如果缓存大小不

稳定，最终置换算法会发挥作用。获取所有条目保留时间的快照，查看快照中哪些条目是“热”的（频繁使用），哪些是“冷”的（较少使用）。你往往能够看出某种模式，例如可能发现 80% 的缓存条目最近被使用过，而其余条目明显更旧。这 80% 近似地代表了对性能改进做出贡献的缓存数量。

还有一种方法——估算缓存命中率。如果我们考虑为运行系统添加缓存，可以收集数据日志并估算缓存命中率。例如，可以收集 24 小时的查询日志并计算重复请求数。重复请求与总查询次数的比率是在添加缓存的情况下，缓存命中率的很好预测器。这种方法假定有限的缓存永远不会过期。如果在这种理论上最佳的条件下缓存命中率很低，我们就知道缓存没有帮助。但是，如果有重复的查询，累计查询响应的总尺寸，就能够很好地估计缓存的大小。

从真实或者基准系统进行计量的问题在于，它们要求系统存在。在设计系统时，重要的是能够对所需的缓存大小做出合理的预测。

可以使用缓存模拟器改进估算。这些工具可以用于提供假设（what-if）分析，以确定最小缓存尺寸。一旦部署缓存，应该监控缓存命中率。缓存的大小可以定期重新评估，在必要时增加缓存以改进性能。

5.5 数据分片

分片是一种灵活、可伸缩、有弹性的数据库分段（ z 轴）方法。它根据数据库键值的散列值分割数据库。散列函数是一种算法，将不同长度的数据映射为固定长度的值。结果从概率上可以视为唯一的，例如，MD5 算法对任何输入返回一个 128 位的数字。因为共有 340 282 366 920 938 463 463 374 607 431 768 211 456 种可能的组合，两个输入产生相同散列的可能性非常小。即使数据中很小的改变也会造成散列的巨大变化。Jennifer 的 MD5 散列是 e1f6a14cd07069692017b53a8ae881f6，而 Gennifer 的 MD5 散列则是 1e49bbe95b90646dca5c46a8d8368dab。

为了将数据库划分为两个分片，生成键值的散列，并将散列为偶数的键值保存在一个数据库中，散列为奇数的键值保存在另一个数据库中。如果要将数据库划分为 4 个分片，可以根据键值散列除以 4 的余数（即散列 $\bmod 4$ ）分割数据库。余数为 0、1、2 或者 3，将指明在 4 个分片中的哪个分片保存键值。因为散列值在分片中是随机分布的，每个分片自动保存的键值数量几乎相同。这种模式称为分布式散列表（Distributed Hash Table, DHT），因为它将数据分布到许多机器上，并用散列决定数据的存储位置。

2 的幂次

我们使用 2 的幂次优化散列—分片映射过程。当你要求得散列除以 2^n 的余数时，只需要查看散列的最后 n 位。这是很快的操作，比用不是 2 的幂次的数字获得模数要快得多。

分片可以复制到多台机器以改善性能。利用这种方法,每个副本处理以该分片为目标的一部分查询。复制还可以提供更好的可用性。如果任何分片都在多台机器上保存,任何机器崩溃或者停机维护时,其他副本将继续服务请求。

存储更多数据时,分片可能超出机器的容量。数据库可以分割为两倍数量的分片。分片数量加倍,旧的数据也划分到新的分片中。求取每个键值的散列以确定键值应该留在当前分片,还是属于新的分片。实时执行这一步骤的系统使用复杂的算法管理扩展期间接收到的查询。

要求分段数量为2的幂次相当不灵活。例如,从一个分片变成两个只需要添加一台机器,这很容易采购。但是,随着系统的增长,你可能发现自己需要从32台机器变成64台机器,这是相当大的采购项目。下一跳的幅度是当前的两倍,如果新机器更强大,额外的能力在较小的机器被淘汰之前都会浪费。而且,虽然键值的数量均匀分布在不同分片中,但是每个键值可能不会保存完全相同的数据量。因此,某台机器可能包含1/4的键值,但是超出了该机器所能容纳的数据量。必须根据最大分片增加分片数量,最大分片可能远远大于最小分片。

对这些问题的解决方案是创建更多较小的分片,并在每台机器上保存多个分片。现在可以改变某台机器上的分片数量,以弥补不均匀的分片尺寸和不同的机器规格。例如,可以用2的更大幂次划分散列值,产生8192个“篮子”。将这些篮子划分到所需数量的机器上。例如,如果有3台机器,其中一台的容量是其他两台的2倍,则较大的机器可以保存在较大服务器上落在0~4095号篮子的键值(共4096个),而在第二和第三台机器上分别保存4096~6143和6144~8191(每台机器2048个键值)号篮子的键值。

当更强大的新硬件可用时,可以在一台机器上放入更多分片。更多分片意味着更多的查询会被导向该机器。因此,需要更多的CPU和网络带宽。当机器保存最大数量的分片时,CPU和网络带宽可能耗尽,性能遭到严重影响。新硬件型号应该进行基准测试,确定可用容量之后才投入使用。理想的情况下,CPU、分片存储量和网络带宽应该同时达到极限。

如果分片用于读/写数据库,每次写入操作更新对应的分片。副本必须保持最新,遵循第1章中讨论过的CAP原则。

在现实中,分片常常用于分布只读的信息库。例如,搜索引擎收集数据并进行索引,产生只读数据库的分片。然后,这些分片需要被分布到每个搜索群集副本上。分布可能相当复杂,传输更新的分片可能花费很长的时间。如果复制数据覆盖旧数据,在更新完成之前,服务器无法响应对该分片的请求。如果传输由于某种原因失败,该分片将无法使用。作为替代,可以在分片传输时为其留出存储空间供临时使用。当然,这意味着没有更新时有未使用的空间。为了消除这种浪费,可以停止通告该分片位于此机器上,以便由其他副本代为处理请求。现在分片可以升级,在没有接收任何新请求之前不通告的过程称作“排空”(Draining)。运营人员必须意识到,在分片排空时,副本减少了一个——性能可能受到

影响。从全局上协调分片升级，保证任何时刻都存在足够的副本，以维护可靠性、达到性能目标，是很重要的。

5.6 线程处理

为了获得更好的伸缩性，可以不同的方式处理数据。每次处理一个请求的方法有局限性。线程处理是通过同时处理许多请求来改善系统吞吐率的一种技术。

线程处理是现代操作系统使用的一种技术，允许指令序列独立执行。线程是进程的子集，在线程之间切换操作通常比在进程之间切换更快。我们使用线程获得对复杂算法处理的更细粒度控制。

在单线程处理中，我们接收一个查询、处理查询、发送结果，然后读取下一个查询，很简单直接，这样做的第一个缺点是某个冗长的请求将拖延后面的请求。这就像只想购买一盒口香糖，却排在购物车装得满满的顾客后面。在这种所谓的队头堵塞情况下，队列的开头被大请求所阻塞。结果是本可以快速服务的请求却得到很高的延迟。

单线程的第二个缺点是在大批请求中有些请求会被丢弃。内核将排队入站连接，同时等待程序取得下一个连接并进行处理。因为内核限制允许的等待连接数量，所以如果有大批新链接涌入，有些就会被丢弃。

最后，在多核处理器中，单个线程将绑定到单个 CPU，而使其他核心空闲。多线程代码能够利用所有核心，从而最大限度地利用机器。

在多线程处理中，**主线程**（Main Thread）接收新的请求。对于每个请求，它创建一个新线程（**工作者线程**）完成实际工作并发送响应。由于线程创建很快，主线程能够应付大批新请求而不会丢弃它们。由于请求并行处理，利用了多个 CPU，队头阻塞得以减少或者消除，从而增大了吞吐率。

话虽如此，多线程较难实现。如果多个线程必须访问内存中的相同资源，就需要加锁或信号标志（信号量）以避免资源冲突。加锁机制很复杂，容易出错。

根据 RAM 和 CPU 内核的限制，机器可以处理的线程数量有限。如果任何一个核心超载，该核心的性能将很快下降。连接洪泛仍然可能造成请求丢弃，但是可以处理的连接数量增加了。

5.7 队列

以不同方式处理数据获得更好伸缩性的另一种途径是**队列**（Queuing）。队列（Queue）是保存请求直到软件准备好处理的一种数据结构。大部分队列按照接收的顺序释放元素，称为先进先出（FIFO）处理。

和多线程类似，队列中也有主线程和工作者线程。主线程收集请求并将其放入队列。

工作者线程的数量通常是固定的，每个工作者线程从队列中获得请求、处理、发送响应，然后取下一个项目。这种工作流程称为“队列馈送”。

5.7.1 优点

队列和多线程有许多共同的优缺点。与此同时，与简单的多线程相比，它有几项优势。

使用队列时，机器超载的可能性较小，因为工作者进程的数量是恒定不变的。保持相同线程服务多个请求也是一种优势，这避免了新线程创建相关的开销。线程创建是轻量级的任务，但是大规模进行时开销也可能累积。

队列模型的另一个好处是更容易实现优先级方案。高优先级请求可以进入队头。根据所要实施的优先级方案类型，有大量的队列算法可用。公平队列（Fair Queuing）算法避免低优先级请求因大量高优先级请求而“饿死”。其他算法则动态改变优先级，使突发或者间歇性的流量不会导致系统超载，或者使其他优先级的请求无法得到处理。

5.7.2 变种

队列模型的变种可能优化性能，它们常常有能力减小或者增大线程数量，这可能根据需求自动进行，也可能人工控制。另一变种是定期杀死和重建线程，使其保持“新鲜”。这缓解了内存泄漏和其他问题，还能隐藏线程，使它们难以找到。

最后，使用进程代替线程是常见的做法。进程创建可能代价很高，但是工作者进程固定存在意味着，可以付出一次开销就得到使用进程的好处。进程可以完成线程无法完成的任务，因为它们有自己的地址空间、内存和打开文件表。进程是自我隔离的，它们的损坏不会影响到其他进程，而表现不佳的线程可能对其他线程造成不利影响。

以进程实现的队列例子之一是 Apache Web 服务器的 Prefork 处理模块。在启动时，Apache 分出一定数量的子进程。请求由主进程分配给子进程。因为子进程已经存在，隐藏了冗长的进程创建，从而更快地处理请求。进程被配置为在每 n 个请求之后死亡并刷新，从而避免了内存泄漏。使用的子进程数量可以动态调整。

5.8 内容分发网络

内容分发网络（Content Delivery Network, CDN）是一种更高效的代替服务交付内容（网页、图像、视频）的 Web 加速服务。CDN 在世界各地的服务器上缓存内容。内容请求从最靠近用户的缓存提供服务，使用定位技术识别请求浏览器的网络位置。

CDN 不将所有内容复制到所有缓存，而是注意使用趋势并确定缓存某些内容的位置。例如，发现某个特定图像在德国大量使用，CDN 可能将该客户的所有图像复制到德国的服务器。这些图像可能代替最近没有被访问的缓存图像。

CDN 有超大规模的快速互联网连接，它们的互联网带宽比大部分网站更大。

CDN 常常将缓存服务器放在 ISP 的数据中心——主机托管，因而减少了 ISP 之间的流量。考虑到 ISP 对此流量收费，流量的减少很快就能补偿为此所做的投资。

通常，网页中间的一个图像可能来自于同一服务器上的某个 URL。但是，图像很少改变。使用 CDN 的网站可以将图像的拷贝上传到 CDN，从指向 CDN 服务器的 URL 中提供该图像。然后，网站使用 CDN 的 URL 引用图像，当用户加载网页时，图像来自 CDN 服务器。CDN 使用各种技术以比网站更快的速度交付图像。

将内容上传到 CDN 是自动进行的。网页像平常那样提供内容，指向该内容的链接称为原生 URL。为了激活 CDN，用指向 CDN 服务器的 URL 代替生成的 HTML 中的原生 URL。URL 对原生 URL 进行编码，如果 CDN 已经缓存了内容，则按照预期提供内容。如果是第一次访问特定内容，CDN 从原生 URL 加载内容、缓存并向请求者提供。在 CDN URL 中编码原生 URL 的思路十分巧妙，意味着无须执行任何特殊的步骤即可将内容上传到 CDN。

最佳实践是使用一个标志或者软件开关确定系统生成网页时输出原生 URL 还是 CDN URL。有时候，CDN 出现问题，你希望可以简单地切换回 URL。有时候，问题不在 CDN 而在于你所造成的配置错误。无数营销材料吹嘘 CDN 产品的可靠性，但是无法让你免于这种情况。而且，新网站在测试阶段时，你可能不希望使用 CDN，特别是测试尚不应该暴露于世界的新的秘密功能时。最后，有了这些开关，就很容易在不同的 CDN 供应商之间切换。

对于小网站，CDN 是极佳的选择。但是，一旦网站变得极大，运营自己的私有 CDN 可能更有成本效益。Google 最初使用第三方 CDN 改善性能，当它处于年轻时代，这一做法使其正常运营时间升级了一个数量级。随着 Google 成长，它在全球建立了自己的数据中心，到这个阶段，Google 构建了自己的私有 CDN，将正常运营时间又升级了一个数量级。

CDN 在 20 世纪 90 年代末才出现，那个时候，它们的焦点是图像和 HTML 文件的静态内容交付。下一代 CDN 产品增加了视频托管。过去文件的大小有限，但是视频托管要求 CDN 能够处理更大的内容，并且处理与视频快速变化相关的协议。因此，当前的 CDN 产品可以作为代理。所有请求通过 CDN，由其作为中介执行缓存服务，重写 HTML 使之更加高效，并支持其他功能。

现在，CDN 在价格、地理位置覆盖和不断增长的新功能上展开竞争，有些 CDN 专长于世界的某个部分，为用户仅集中于这一部分的网站提供更低的价格，或者提供特殊的服务，比如获得许可（也就是说，为难以通过艰难复杂的流量审查要求的客户提供帮助）。

新功能包括提供动态内容、为移动设备提供不同内容和安全服务的能力。HTTPS（加密 HTTP）服务的管理可能既复杂又困难，有些 CDN 可以替你处理 HTTPS 连接，使你免于管理这种复杂性。Web 服务器和 CDN 之间的连接可以使用更易于管理的传输机制，或者完全不加密。

5.9 小结

大部分伸缩性方法都可以归入 AKF 伸缩立方体的一个轴。x 轴（水平伸缩）是力量倍增器，克隆系统或者增大其容量以实现更高的性能。y 轴（垂直伸缩）通过按照事务的类型或者范围进行隔离实现伸缩性，例如为读查询使用只读数据库副本，仅让写操作进入主数据库。最后，z 轴（基于查找的伸缩性）将数据分割到不同服务器，使工作负载按照数据使用率或者物理位置分布。

分片在多个服务器上放置数据库（行）的水平分区扩展大型数据库，获得了更小索引和分布式查询的好处。将分片复制到更多服务器，可以实现速度和可靠性上的好处，但是需要付出数据新鲜度的代价。

另一种数据存取优化方法是缓存，这是快速 / 昂贵媒体上相对小型的数据存储。缓存汇聚最近请求的数据，在请求不包含在缓存中的数据时对自身进行更新。后续的查询可以直接从缓存读取，实现总体性能改善。

线程处理和队列为我们提供了处理大量请求、将其汇聚为可单独服务的结构的一种工具。内容分发网络提供了 Web 加速服务，通常采用的方法是在更靠近用户的地方缓存内容。

练习

1. 什么是伸缩性？
2. 为受到 CPU 限制的服务实现伸缩性有哪些选择？
3. 为存储需求增长的服务实现伸缩性有哪些选择？
4. 图 1.10 中的数据因为硬件每年都变得更便宜而过时。更新当年的图表，哪些项目变化最小，哪些变化最大？
5. 按照比例重写图 1.10 中的数据。如果从主存读取花费 1 s，其他操作花费多长时间？将你的答案画成日历或者太阳系的样子可以得到额外的加分。
6. 取图 1.10 中的数据表格并添加一列，标识每个项目的成本。以相同的单位对成本进行比例调整，例如，1 TB RAM、1 TB 磁盘和 1 TB L1 缓存的成本。添加另一列表示性价比。
7. 描述不同伸缩性技术的理论模型是什么？
8. 如何知道何时需要伸缩？
9. 最常见的伸缩性技术有哪些？它们的工作原理是什么？何时最适合使用它们？
10. 哪些伸缩性技术还能改善弹性？
11. 描述你的环境如何使用 CDN 或者研究可能的使用方法。
12. 研究 Amdahl 法则，解释它与 AKF 伸缩立方体的关联。

Chapter 6 第6章

弹性设计模式

成功并非终点，失败也不是末日：最重要的是继续前进的勇气。

——温斯顿·丘吉尔

弹性是系统以建设性的方式处理故障的能力。弹性系统检测故障并绕开，非弹性系统在面对故障时会崩溃。本章介绍基于软件的弹性以及最常使用的技术文档。

弹性之所以重要，是因为没人会前往崩溃的网站。硬件失效——生活中的现实。你可以购买全世界最可靠、最昂贵的硬件，但是仍然会出现很多故障。在规模足够大的系统中，即使故障率只有百万分之一，问题也会每天发生。

在典型 Google 数据中心运营的第一年，会发生 5 次机架范围的运行中断。3 次需要将处理转移到所连接机器之外的大型路由故障，以及 8 次计划内的网络维护窗口——其中半数会导致 30 分钟的随机连接丢失。与此同时，所有磁盘中有 1%~5% 损坏，每台机器至少崩溃 2 次（故障率 2%~4%）(Dean, 2009)。

前面讨论过的优雅降级意味着，设计软件时通过提供精简的功能，在故障或者高负载期间存活下来。例如，在互联网连接故障或者超载时，电影流服务可能自动降低视频分辨率以节约带宽。另一种策略是纵深防御 (Defense in Depth)，也就是说设计的所有层次检测故障并做出反应。这包括单个进程的小型故障和整个数据中心的大型故障。

实现可靠性较为老旧、传统的一种策略是在可能发生故障的每个地方降低其概率。使用最好的服务器和最好的网络设备，并将其放在最可靠的数据库。采用这种策略时仍然可能出现运行中断，但是很少见。这是最昂贵的策略。另一种策略是执行依赖性分析，验证每个系统依赖的是高质量部件。制造商计算其制造的部件的可靠性并发布平均无故障时间 (Mean Time between Failure, MTBF) 等级。通过分析系统中的相互依赖性，可以预测整个

系统的 MTBF。系统的 MTBF 只能达到最低 MTBF 部件的水平。

必知术语

运行中断：用户可见的服务缺失。

故障：系统、子系统或者组件停止工作。

失效：与“故障”可以互换使用。

服务器：提供功能或者 API 的软件（不是硬件部件）。

服务：由一个或者多个服务器组成的用户可见系统或产品。

机器：虚拟机或者物理机。

QPS：每秒查询数。通常是每秒 Web 访问次数或者 API 调用次数。

这种策略是预测性的，预测的是故障的可能性或者系统的可靠性。

弹性系统比预测性策略更近一步。假定故障将会发生，我们构建智能做出反应的系统，使得整个系统能够存活并继续提供服务。换言之，弹性系统对故障的反应很灵敏。它们并不通过更好的硬件避免故障，或者以人力（以及道歉）对其做出反应，而是采取主动态势，部署预期故障并能够存活的机制。

弹性系统消除组件故障与用户可见运行中断之间的耦合。在传统计算中，只要组件发生故障，就会有用户可见的运行中断。当我们构建可存活系统（survivable system）时，这两个概念被解耦。

本章介绍的是设计检测故障并做出变通的系统的各种方法，这也是我们构建可存活系统的方法。这些技术被分为 4 类：物理故障、攻击、人为错误和意外的负载。

6.1 软件弹性胜过硬件可靠性

可以通过选择更好的硬件或者更好的软件来构建可靠的系统。更好的硬件意味着专用的 CPU、组件和存储系统。更好的软件意味着为系统增加智能，使其检测故障并绕开它们。

软件解决方案得到青睐有许多原因。首先，它们更经济。软件编写好之后，可以应用到许多服务和机器上，而没有附加成本（假设此软件是内部开发、开源或者不受机器许可限制）。软件的适应性也比硬件更强，它更容易修复、更新和替换。和硬件升级不同，软件升级可以自动化。因此，软件常常被替换。新功能可以更快、更频繁地推出，试验也更容易。软件随着时间的推移而变得更加强大：缺陷得到修复，罕见的边缘情况得到更好处理。Spolsky（2004）的论文《Things You Should Never Do》给出了许多例子。

相比之下，使用更好的硬件较为昂贵。硬件的初始采购价格更高，更可靠的 CPU、组件和存储系统比常见部件要昂贵的多。这种策略较为昂贵还有一个原因：在成长期内，每台机器都需要支付额外的费用。硬件升级在每台机器上需要付出硬件本身、安装工作、资产折旧和旧部件处置等成本。因为硬件设计所花费的时间更长，所以升级的频率较低，新

部件的试验也更困难。硬件越陈旧，就越脆弱，越经常出现故障。

6.2 所有东西最终都会失灵

故障是每个环境的一部分，它们可能发生在每个级别。例如，故障可能发生在组件级（芯片和其他电器部件）、设备级（硬盘、主板、网卡）和系统级（计算机、网络设备、电源系统）。故障还可能在某个区域发生：机架失去电源、整个数据中心下线、城市和整个地区遭到灾害的袭击。人为因素也可能成为故障的根源，包括打字或者软件缺陷、不小心踢到电源线使其从插座中脱落以及蓄意的恶意攻击等。

6.2.1 分布式系统的 MTBF

大规模系统会放大小问题。在大系统中，“百万分之一”的问题会经常发生。MTBF 为 100 万小时的硬盘在一年内有 1/114 的概率会出现故障，如果你有 10 万块这种硬盘，可以预期每天有两块硬盘出现故障。

2010 年父母家中的 PC 发生一次死机，其起因可能是 CPU 中一个触发概率为千万分之一的缺陷。他们可能咒骂几句，重新启动 PC，就不再想起这件事了。这种缺陷是芯片制造商所难以检测到的。但是，在分布式计算系统中，同一个缺陷可能频繁出现，成为崩溃检测和分析系统中的一个模式。该问题会被报告给供应商，后者会对问题的存在感到错愕、为问题的发现而震惊，并对这样一个问题出现在多代芯片的核心 CPU 设计中而羞愧。供应商也不太可能允许这些细节出现在系统管理的书籍上。

故障集中爆发，会让我们觉得似乎所有的机器联合“捣乱”。在电源中断之后，几个机架的机器试图同时启动，这会使濒临极限的电源无法提供足够的动力，同时驱动数十块磁盘。旧的焊点收缩开裂，导致神秘的故障。来自同一制造批次的组件有类似的“死亡率曲线”，造成突发的大量故障。

在我们对许多潜在故障和失效的讨论中，希望不会吓得大家离开系统管理这一领域！

6.2.2 传统方法

传统软件设想一个完美的无故障世界。这给硬件系统工程师留下了无法完成的任务——交付永不失效的硬件。我们用廉价[独立]冗余磁盘阵列（Redundant Array of Inexpensive [Independent] Disk, RAID）系统假造了这种系统，让软件继续假定磁盘永不失效。我们掩盖了世界充满故障的现实，让软件开发人员继续编写设想无故障完美世界（当然，这并不存在）的软件。

例如，UNIX 应用是在文件读写不会出错的假设下编写的。因此，应用程序写入文件时不检查错误。应用程序进行这些工作是在浪费时间，因为数据块可能在以后才写入磁盘，那时应用程序可能已经退出了。Microsoft Word 也是在运行软件的计算机将会继续运行的假

设下编写的。

夸张的警告

前一段有两处稍作夸大。UNIX 的应用层假设完美的文件系统，但是底层并没有假设完美的磁盘。Microsoft Word 检验文档，使得用户在崩溃时不会丢失数据。但是，在崩溃时用户无法编辑文档。

试图实现这种不可能存在的无故障世界，导致公司花费大量的金钱。高可靠性的 CPU、组件和存储系统明显比常见部件昂贵得多。附录 B 详细介绍这一策略的历史，解释了本章讨论的分布式计算技术的经济效益。

6.2.3 分布式计算方法

和传统方法相比，分布式计算接受组件的故障和失效。它采用务实的方法，将故障作为生活中的现实加以接受。Google Docs（谷歌文档）即使在 Google 的机器失效时仍继续让用户编辑文档；另一台机器接管工作，用户甚至都没有注意到移交工作的发生。

传统计算不遗余力地通过硬件实现可靠性，然后将少量故障作为“正常”情况接受，或者添加检测和绕过故障的智能。如果软件能够绕过故障，在昂贵的硬件上花费金钱就是一种浪费。

有一条流行的保险杆标语“适当饮食、锻炼，人还是会死”。如果硬件不管多贵都会失效，为什么要买最好的呢？为什么多花一倍的价格去追求可靠性？

购买失效的内存

在 Google 发展的早期，它试图了解自身使用智能软件管理不可靠硬件的极限。为此，该公司购买失效的 RAM 芯片，寻找使其可用的途径。

Google 购买了几个 TB 的 RAM，用于运行高故障弹性软件的机器。如果芯片失效，操作系统将标记该区域的 RAM 不可用，并杀死使用这些 RAM 的进程。被杀死的进程将会自动重启，芯片损坏的事实被记录下来，即使重新启动，操作系统也会忽略这些芯片。

结果是，仅因为一个芯片失效，机器没有必要维修。该机器能够继续运行，直到机器容量降低到可用的限度之下。

为了理解接下来发生的事，你必须理解高质量 RAM 芯片和正常质量芯片之间的差别在于所接受的测试数量。RAM 芯片制造后要经过测试，通过最多 QA 测试的芯片被作为“高质量”芯片，以高价格出售。通过标准 QA 测试的芯片被作为正常芯片，以常规价格出售。其他芯片则被丢弃。

Google 的采购人员是令人敬畏的谈判专家。Google 已经通过购买常规质量的芯片，依靠自己编写的定制软件绕过故障，节约了金钱。有一天，采购部门询问是否可以购买被丢弃的芯片。制造商之前从未接到这样的请求，他们愿意出售有缺陷的芯片，换回少

数资金。

对 Google 的需求，“失效”的 RAM 芯片工作得很完美。有些从一开始就无法工作，其他一些则很快就真正失效。但是，服务仍然保持运行。

Google 最终停止采用这一做法，但是许多年间该公司都以低于竞争者的价格构建具备海量 RAM 的服务器。当你的业务是对每条广告索价几美分时，几美元的节约就是一个巨大的优势！

6.3 通过备用容量实现弹性

增加弹性的一般策略是采用可以独立失效的冗余容量单元。检测出故障时，这些单元从服务中删除。系统总容量减少，但是仍然可以运行。这意味着，系统从一开始就必须具备**备用容量**（Spare Capacity）。

我们以一个提供网站上显示静态图像的 Web 服务器作为例子。这种服务器很容易复制，因为内容不经常变化。例如，我们可以构建多个此类服务器，并在其中平衡负载（负载均衡器的工作原理在 4.2.1 节中讨论）。我们将这些服务器称为**副本**，因为每台服务器复制相同的服务。它们作为副本都能够响应相同的查询，并得出等价的结果。在这种情况下，用相同的 URL 访问同一图像。

假定每个副本可以处理 100QPS，在高峰期接受 300QPS，需要 3 个服务器以提供 300QPS 容量，还需要一个附加副本提供失效副本存活所需的备用容量。任何一个副本故障时，该副本被自动撤出服务，负载此时平衡到存活的 3 个副本上。系统总容量降低到 300QPS，这是足够的。

我们将这种方案称为 **N+M 冗余**，这种系统需要 N 个单元提供容量，具备 M 个单元的额外容量。单元是提供服务的最小离散系统。正如上面的例子，当我们希望表示有足够应付一处故障的备用容量时，使用术语 **N+1 冗余**。如果我们添加第 5 台服务器，系统能够在两个副本同时故障时存活，可以描述为 **N+2 冗余**。

如果我们采用 3+1 冗余且发生一系列故障时会怎么样呢？在第一个故障发生后，系统可以描述为 3+0，仍然运行但是已经没有冗余。第二个故障将造成系统**超出配额**，也就是说，可用容量少于所需容量。

继续前一个例子，当有两个副本失效时，容量为 200QPS。该系统现在超配比例为 3:2——在需要 3 个副本的情况下有两个副本存在。如果我们幸运，这种情况可能发生在用户不多的时段，200QPS 也足够应付。但是，如果运气不好，这种情况发生在高峰时段，余下的两台服务器将面对 300QPS，这将超出设计处理能力。这种超载的处理在后面的 6.7.1 节中介绍。

6.3.1 需要多少备用容量

备用容量就像保险策略：现在支付的用于准备未来发生的麻烦（你不希望发生）的费用。购买保险而不需要用到，比在需要保险时却没有事先购买要好。话虽如此，购买过多的保险是一种浪费而不是好的做法。选择容量单位粒度可以控制效率。例如，在 1+1 冗余系统中，有 50% 的空闲容量。在 20+1 冗余系统中，备用容量小于 5%。后者的成本效益更高。

选择冗余数量的其他因素是我们提供附加容量的速度，以及在此期间发生第二次故障的可能性。维修或者替换故障容量的时间称作平均修复时间（Mean Time to Repair, MTTR）。在此期间发生运行中断的概率是平均无故障时间的倒数。在维修窗口期发生第二次故障的概率为 $MTTR/MTBF \times 100$ 。

如果第二次故障意味着数据丢失，第二次故障的概率就成为决定应有备用容量数量的重要因素。

假定修复故障单元需要花费 1 周（168 小时），而 MTBF 为 10 万小时，则第二次故障的概率为 $168/1\,000\,000 \times 100 = 1.7\%$ ，约为 1/60。

现在假设 MTBF 为 2 周（336 小时）。在这种情况下，第二次故障的概率为 $168/336 \times 100 = 50\%$ ，也就是 1/2——和丢硬币猜正反面的概率相同。添加附加的副本成为当务之急。

MTTR 是一些因素的函数。需要重启的死锁进程 MTTR 很短。损坏的硬件组件可能只需要几分钟就可以更换，但是如果服务器在 9000 英里^①之外的数据中心，到达那里可能需花费一个月。备件需要订购、运输和交付。即使磁盘能在故障发生后的几分钟内更换，如果采用的是 RAID 配置，需要漫长的重建过程，在完成之前系统仍然是 N+0 的配置。

如果这些计算让你头疼，可以使用简单的经验法则：N+1 是服务的最低配置。如果在修复第一个故障期间发生第二次运行中断，则需要 N+2。

数字计算机或开或关，我们也可以这样看待服务：它或者上线或者下线。当我们通过复制实现弹性，服务就更像模拟设备：它可能为开、关或者两者间的任何状态。我们不再监控服务以确定其上线或者下线，而是监控系统中的容量，确定是否应该添加更多容量。这改变了对系统及运营的想法，我们不再因为一台机器下线而在半夜被叫醒，而只在仪表指针靠近危险区的时候才得到警报。

6.3.2 负载平衡与热备份的对比

在前面的例子中，副本是负载平衡的：所有副本都是活动的，（近似）平等地共享工作负载，备用容量也（近似）相等。另一种策略是采用主辅副本。在这种方法中，主副本接受整个工作负载，而辅副本做好准备在任何时候接管工作。这有时被称作热备份（Hot Spare 或者 Hot Standby）策略，因为备用设备连接到系统运行（热），可以立即切换到操作中。这

① 1 英里 = 1.609 千米

种方法也被称为主动—被动或者主—从配对。辅副本往往有多个，因为只有一个主副本，这种配置是 1+M 配置。

有时候，“主动—主动”或者“主—主”配对指的是两个负载均衡的副本。“主动—主动”更常用在网络链接上。“主—主”较常用于数据库领域和两者紧密耦合的情况。

6.4 故障域

故障域 (Failure Domain) 是一个有限范围，故障的影响不会超出这一范围。例如，当车辆在公路上出现故障，其故障不会使整条公路无法使用。故障的影响限制在其故障域内。

家用断路器箱中保险丝的故障域是电路所覆盖的一两个房间。如果电源线被切断，故障域影响许多房屋，也可能影响整个街区。电网的故障域可能是由其供电的城市、地区或者县（这就是一些数据中心在选择位置时采取策略，使其能够连接到两个电网的原因）。

故障域可以事先规定——设计目标或者需求。你可能规划两组服务器各有自己的故障域，然后设计系统以实现这个目标，确保它们自身的故障域相互独立。每组服务器可能处于不同机架、不同供电线路等。它们是否处于不同数据中心，取决于故障域目标范围。

故障域也可能是描述性的。我们往往发现自己正在探索一个系统，试图确定（或者逆向工程）结果故障域。由于某一机器发生故障，服务器可能已经被临时转移到另一个机架上的备用机器。我们探索这一转移的影响，以确定新的故障域。

故障域的确定是在某个特定范围之内，或者在关于我们所愿意考虑的运行中断范围大小的假设下完成的。例如，我们可能将场外备份放在 1000 千米之外，假定影响两座离得很远的建筑物的运行中断是可以接受的风险，或者那么大的灾难意味着我们还要考虑其他问题。

不统一的故障域增大运行中断的影响

某公司有许多大型数据中心，采用 6 个机架为一组，共享电源总线的架构。每 8 个机架由一个网络子系统提供网络连接性。网络子系统从每个组的第一个机架取得电源。

如果电源总线需要关闭维护，将造成直接连接到该总线的 6 个机架中断运行，此外，如果其他机架恰好处于从受影响的机架上获取电源的网络子系统中，它们会失去网络连接而中断运行。这将故障域扩展到 13 个机架。许多用户会觉得在维修不直接影响他们时也遭受这样的待遇是不公平的。

其他一些不统一的故障域与散热及群集管理器所管理的机器相关，结果是，这些不统一不仅造成不便，而且成为影响系统可用性的重要因素。

最终，该公司采用了新的数据中心设计，将所有物理故障域统一到某个“公倍数”。在某些情况下，这意味着和供应商一起定制设计。旧的数据中心最终花费巨资改造为新的设计。

我们常常听说数据中心在电源、网络和其他因素上达到完美统一，只有某种意外的不

协调情况会导致重大的运行中断。例如，考虑一个有 10 个域的数据中心，每个域都有独立的电源、散热和网络系统。假定该建筑物到外界有两路连接，相关的设备根据连接进入建筑物的位置布置。如果两个包含这些连接的域中出现散热故障，全部 10 个域就会突然无法连接到外界。

6.5 软件故障

只要有软件的地方，就曾经出现过软件缺陷。长期运行的软件可能意外死机。软件可能挂起、失去响应。由于这些原因，软件也需要弹性特征。

6.5.1 软件崩溃

系统中的常见故障之一是软件崩溃，或者过早退出。软件崩溃有多种原因和应对方法。服务器软件通常是准备长期运行的，例如，除非更改配置需要重启或者服务退役，否则提供特定 API 的服务器应该是永远运行的。

软件崩溃有两类：

- ❑ **常规崩溃 (Regular Crash)** 发生在软件进行某些操作系统禁止的工作时。例如，由于某个软件缺陷，程序试图写入操作系统标记为只读的内存，操作系统检测到这一现象并杀死该进程。
- ❑ **恐慌 (Panic)** 发生在软件本身检测到某种错误，确定最好的方法是退出时。例如，软件可能检测到不应该存在且无法更正的情况。该软件的作者可能确定这种情况下最安全的做法就是退出。例如，如果内部数据结构被破坏，没有安全的改正手段，最好是立即停止工作，而不是用不好的数据继续。恐慌本质上是有意造成的崩溃。

自动重启和升级

处理软件崩溃的最简单方法是重启软件。有时候，问题是暂时性的，重启就足以修复它。这种重启应该是自动化的。对于几千台服务器，人工检查进程以确定它们是否退出并在必要时重启是低效的。处理这一任务的程序称作**进程监视器 (Process Watcher)**。

但是，重启停止的进程并不像看上去那么简单。如果它立刻再次崩溃，我们需要采取其他措施，否则，我们将浪费 CPU 时间而于事无补。进程监视器通常会检测到进程已经在 y 分钟内启动 x 次，并认为这种行为导致问题升级。升级不仅涉及进程的重启，而且还要将问题报告给某个人。这一阈值的例子之一是在一分钟内重启某个进程 5 次。

频率较低的重启往往是其他问题的信号。每小时重启一次不会触发警报，但是应该加以调查。这种较慢的重启问题往往由监控系统而非进程监视器检测。

自动化崩溃数据收集和分析

每次崩溃都应该记入日志。崩溃通常在一个崩溃报告中留下许多信息。崩溃报告包括

进程死机时的 RAM 和 CPU 使用量等统计数字，以及问题发生时函数调用的回溯及所执行代码行等详细信息。崩溃期间往往会生成一个核心转储（Coredump）文件——包含进程内存内容的文件。开发人员使用该文件协助调试。

崩溃报告的自动化收集和存储很有用，因为如果没有很快收集，这些信息可能会丢失，这些信息可能会被删除，或者所在机器可能消失。人工收集这种信息很不方便，但是对于自动化来说很容易，尤其是在具有数百台机器、数十万个进程的系统上。集中存储崩溃报告可以实现数据挖掘和分析。简单的分析结果（如哪些系统最常崩溃）可能成为很有用的工程指标。更复杂的分析可能发现公共软件库、操作系统、硬件甚至特定芯片中的缺陷。

6.5.2 软件挂起

有时候，当软件出现问题时没有崩溃而是挂起，或者进入无限循环。

检测挂起的策略之一是监控服务器并检测是否已经停止处理请求。我们可以被动地观察请求计数或者主动发送请求测试系统，验证响应在一定时间内生成。这些主动请求被称作 Ping，设计成轻量级的请求，只验证基本的功能。

如果 Ping 以指定的速度周期性发送，用于检测挂起和崩溃，它们被称作心跳请求（Heartbeat Requests）。检测到挂起时，可以生成错误、发出警报或者尝试重启服务。如果服务器是负载均衡器后许多副本中的一个，可以从负载均衡循环中删除该服务器，调查问题，而不是简单重启。有时候，添加一个新副本比将维修好的副本返回工作状态的工作量大得多。例如，在 Google 文件系统中，添加到系统的新副本可能需要复制 TB 级的文件。这可能淹没网络。修复挂起的副本并使其返回服务中只会造成现有数据的重新校验，这是较为轻量级的任务。

另一种处理软件挂起的技术称作监视计时器（Watchdog Timer）。硬件时钟连续递增一个计数器，如果计数器超过某个值，硬件子系统将检测到并重新启动系统。运行于系统上的软件在任何成功操作之后将计数器重置为 0。如果软件挂起，重置将停止，系统很快就会重启。只要软件保持运行，计数器就会频繁地被重置，避免重新启动。

监视计时器最常用于操作系统内核和嵌入式系统。在具有合适硬件的系统上启用 Linux 内核监视计时器，可以减少在内核挂起时物理访问机器的需求，或者避免购买昂贵的远程电源控制系统的需求。

和崩溃类似，挂起应该记入日志并进行分析。频繁挂起是硬件问题、加锁问题和其他缺陷的标志，这些问题应该在造成大的破坏之前加以修复。

6.5.3 死查询

有时候，特定的 API 调用或者查询会使用未经测试的代码路径，导致崩溃、长时间延迟或者无限循环。我们将这种查询称为死查询（Query of Death），因为它会杀死服务。

当用户发现流行网站上的死查询时，他们会告诉所有的朋友，很快互联网上的大部分

人也会试图了解崩溃网站的情况。公司越有名，消息传播得越快。

最好的修复手段就是消除导致这种问题的缺陷。不幸的是，修复代码和推出新发行版本需要很长时间。同时需要一个快速的修复手段。

广泛使用的一种策略是建立一个容易更新的禁止查询列表（Banned Query List）并传达给所有前端。前端自动拒绝列表中的任何查询。

但是，这种解决方案仍然需要人工干预，需要一种更为自动化的机制，尤其是在查询有很大的扇出量时。例如，假定这个查询接收后发送给 1000 台其他服务器，每个服务器容纳数据库的千分之一。死查询将会杀死 1000 台服务器，以及正在进行的所有其他查询。

Dean 和 Barroso（2013）描述了在 Google 率先推出的预防性措施，称作“金丝雀请求（Canary Requests）”。在人们正常会向数千台叶子服务器发送相同请求的情况下，使用这种方法的系统向一两台叶子服务器发送查询。这些请求就是“金丝雀请求”，只有金丝雀请求的响应在合理的时间内收到，查询才发送给其他服务器。如果叶子服务器在金丝雀请求处理期间崩溃或者挂起，系统将请求标记为“潜在危险请求”，不向其他叶子服务器发送，以避免进一步的崩溃。利用这种技术，Google 在面临难以预测的编程问题和恶意拒绝服务攻击的情况下实现了一定的健壮性。

6.6 物理故障

分布式系统在面对物理故障时也必须具有弹性。分布式系统中使用的物理设备可能在许多级别上失效。物理故障的范围从最小的电子元件到一个国家的电网。通过在每个级别上实现冗余性提供弹性很昂贵，伸缩性也难以实现。需要一种策略，在不增加过多成本的情况下提供硬件故障弹性。

6.6.1 部件和组件

许多计算机组件都可能失效。可监控利用率的部件（如 CPU、RAM、磁盘和网卡）可能失效，支持组件（如风扇、电源、电池和主板）也可能失效。

历史上，当 CPU 出现故障时，整台机器都不能使用。但是，现在多处理器计算机相当常见，所以只要有一个处理器仍然正常工作，机器就很有可能存活。如果机器已经在这方面实现了弹性，我们必须监控 $N+0$ 的情况。

RAM

RAM 常常因为奇怪的原因失效。有时候，轻微电源浪涌就可能影响 RAM。其他时候，因为来自另一个星系的宇宙射线恰好穿越它，某个位的值会出现反转。这是真的！

许多内存系统在每个字节中存储一个附加位（校验位）使它们可以检测错误，或者使用两个附加位（纠错码或者 ECC 内存），使其可以进行错误校正。这种做法增加了成本，而且

会降低可靠性，因为位数增加了 25%，因而 MTTF 降低了 25%。（但是大部分这类故障现在可以在不为人知的情况下校正。故障仍然发生，但是可以通过监控系统检测。如果故障持续，则需要更换该组件。）

写入校验位时，系统计算字节中为 1 的位有几个，如果总数为偶数则在校验位中存入 0，如果总数为奇数则存入 1。每次读取内存时，检查校验位并向操作系统报告不匹配的现象。这足以检测所有一位错误或者无法保持奇偶性的多位错。ECC 内存使用两个附加位和海明码算法，可以改正一位错并检测多位错。

内存中的值未读出的时间越长、系统中的内存越多，两个或者更多位错误的可能性就越大。

不使用任何校验或者 ECC 位可以节约金钱——低端芯片组常用的方法——但是所有软件都必须有自己的校验和及错误校正，这很慢且代价很高，你或者你的开发人员可能不愿意这么做。那么，就把钱花在 ECC 上。

磁盘

磁盘的故障往往是因为拥有活动部件，固态硬盘（SSD）没有活动部件，它们的损坏是因为每个数据块只能写入一定的次数。

通常的解决方案是使用 RAID 1 或者更高的级别，实现 N+1 或更好的冗余性。但是，RAID 系统成本很高，内部固件往往因为不中断服务就难以配置而令人沮丧（在此不包含 RAID 级别的完整解释，但是可以在我们的另一本书中找到——《The Practice of System and Network Administration》）。ZFS、BTFS 和 Hadoop HDFS 等文件系统通过提供自己的 RAID 或者类似功能，可靠地保存数据。在这种情况下，不需要硬件 RAID 控制器。

我们建议有策略地使用 RAID 控制器，只在必要的地方部署它们。例如，Apache Hadoop 系统是一种广泛使用的分布式计算环境。Hadoop 群集中的前三台机器是特殊的主服务机器，保存关键配置信息。这些信息没有复制，如果丢失了就难以重建。Hadoop 群集中的其他机器是数据节点，保存数据副本。在这种环境中，通常将 RAID 用于主机器。实施 RAID 的成本是固定的，因为需要 RAID 控制器的机器不超过 3 台。数据节点在需要更多容量时加入，它们在构建时不需要 RAID，因为 Hadoop 在必要时复制数据，检测故障并创建新副本。这种策略采用固定数量的昂贵硬件，而用于扩展系统的节点采用廉价硬件，具有成本优势。

电源

每台机器都有电源（Power Supply），将标准电压转换为计算机需要的水平。电源经常失效。服务器、网络设备和许多其他系统购买时可以带有冗余电源，常见的是 N+1 和 N+2 配置。

和 RAID 一样，最好是策略性地使用冗余电源，当系统本身是一个副本或者在更高级

别上使用其他弹性技术时，就不需要它们。在其他没有冗余性的系统上使用这种电源。

网卡

网卡或者网络连接本身经常失效。N+1 配置可以使用多个链接，这有许多标准，无法一一详述。

有些网卡使用负载平衡，另一些则使用主动—被动模式。有些网卡要求所有近端（机器）连接插入同一个网络接口控制器（NIC）子板。如果两个物理端口共享同一个子板，其中一个的故障可能导致另一个失效。有些网卡要求所有远端（交换机）连接插入相同交换机，而其他则无此限制。后一种方法不仅提供了 NIC 故障的弹性，而且提供了交换机故障的弹性。

许多不同算法能够确定哪些数据包通过哪个物理链接。对于某些算法，数据包可能不按顺序到达，但是所有协议都应该能处理这种情况，其中很多都做得不好。

硬件故障的纵向研究

Google 已经发布了硬件故障的两项纵向研究。这种故障的大部分研究都是在实验室环境下完成的。Google 细致地收集了整个机群的组件故障信息，提供了对实际故障模式的最佳认识。两项研究都值得一读。

《Failure Trends in a Large Disk Drive Population》（大磁盘驱动器群中的故障趋势，Pinheiro、Weber & Barroso 2007）对大硬盘群体进行了多年的分析。作者不寻找温度或者活动水平与磁盘故障的关联，而是发现一次扫描错误被检测之后，驱动器在 60 天内再次出现故障的可能性提高了 39 倍。他们发现了在第一个月或者许多年之后故障发生趋势的“澡盆故障曲线”。

《DRAM Errors in the Wild: A Large-Scale Field Study》（自然环境下的 DRAM 错误：大规模现场研究，Schroeder、Pinheiro & Webe 2009）分析了 2.5 年内数据中心中大型机群的内存错误。这些作者发现，错误率比以前报道的高出几个数量级，主要是硬错误——ECC 能够检测但是不能改正的错误类型。与其他因素相比，温度的影响相对较小。

6.6.2 机器

机器故障通常是组件损坏的结果。如果系统有 N+1 配置的系统，两个故障就会造成机器失效。崩溃的机器常常在关闭电源并重启之后（往往需要一定的延时，让组件排空）恢复正常。这一过程可以自动化，但是重要的是自动化措施能够区分两种情况：机器无法访问和机器将要下线。如果重启电源无法恢复机器，必须对其进行诊断、维修，然后再恢复服务。这些工作中大部分可以自动化，尤其是操作系统的重新安装。这一主题将在 10.4.1 节中更详细介绍。

前面，我们描述了机器在断电之后无法重启的情况。这些问题可以通过定期重新启动提前发现。例如，Google 对机器逐个排空以进行内核升级。这样做的结果是，每台机器在

大约每 3 个月以受控方式重新启动，减少了断电之后发现的意外情况数量。

6.6.3 负载均衡器

不管服务器的故障是因为死机、网络问题还是缺陷，处理这种故障的一种弹性方法是使用副本和某种负载均衡器。

前面描述的用于改善伸缩性的负载均衡器也可以用于改进弹性。但是，使用这种方法改善伸缩性时，增加副本的意图是增加容量。现在，增加的备用容量是我们所不希望使用的保险策略。

使用负载均衡器时，考虑它是用于伸缩、弹性还是两者兼有是很重要的。我们已经观察到一些情况，假定存在负载均衡器就意味着系统可以自动伸缩和具有弹性，事实并非如此。负载均衡器不是魔法，它只是一种有多种不同用途的技术。

伸缩性与弹性的对比

如果我们在两台机器上平衡负载，每台有 40% 的利用率，则任何一台机器死机，另一台机器的利用率将为 80%。在这种情况下，负载均衡器被用于提供弹性。

如果我们在两台机器上平衡负载，每台利用率为 80%，有一台机器下线就没有任何备用容量。如果一台机器死机，剩余的副本将接受所有流量——该机器所能处理的 160%。这台机器将会超载，可能停止工作。两台各达到 80% 利用率的机器等于 $N+0$ 配置。在这种情况下，负载均衡器用于提供伸缩性，而非弹性。

在前面的两个例子中，使用的相同配置：2 台机器和一个负载均衡器。但是在一个例子中实现了弹性，另一个实现了伸缩性。两者之间的不同是利用率或者处理的流量。换言之，当你只有两台服务器，50% 的利用率就等于满负荷。

如果我们在第二个例子中增加第三个副本而流量没有变化，则 300% 的总容量中有 160% 在用。这是 $N+1$ 配置，因为一个副本死机时剩下的副本仍然能够处理负载。在这种情况下，负载均衡器既用于提供伸缩性，又用于提供弹性。

当我们用负载均衡器去赶上容量要求时，它提供的是伸缩性，当用它超越容量需求时，提供的就是弹性。如果利用率升高而我们没有添加更多副本，就可能再无法实现弹性。如果白天的流量很高，晚上的流量很低，我们可能在一天的某个时段得到具有弹性的系统，而在其他时间则无法实现弹性。

负载均衡器弹性

负载均衡器本身可能成为单故障点（Single Point of Failure, SPOF）。往往使用冗余的负载均衡器配对弥补这种不足。

策略之一是简单的故障切换。一个负载均衡器（主）接收所有流量，另一个负载均衡器（辅）发送心跳信息监控主负载均衡器的健康。如果检测到心跳信号丢失，辅负载均衡器接管并成为活动的负载均衡器。任何“在途”TCP 连接将会断开，因为主负载均衡器不知道

它们的存在。

另一种策略是状态型故障切换，这种策略与简单故障切换类似，但是两个负载平衡器交换足够的信息（或者状态），使两者都知道现有的 TCP 连接。结果是这些连接在故障切换时不会丢失。

一对负载平衡器往往用于许多不同的服务，例如，许多不同的网站。所有网站保存在一个负载平衡器上，另一个用于故障切换。使用非状态型负载平衡器时，常见的技巧是将网站的一半放在一个负载平衡器上，另一半放在另一个负载平衡器上。在这种情况下，故障切换时有一半的连接丢失。

硬件或者软件负载平衡器

负载平衡器可能是专门为此任务构建的硬件设施，也可能是运行于标准计算机上的软件程序。硬件设施通常经过精心调整、有丰富的功能。基于软件的负载平衡器更为灵活。

对于较小的服务，负载平衡器软件可能运行于提供服务的机器上。将负载平衡功能推送到机器本身，减少了需要管理的硬件数量。但是，这样负载平衡软件的处理和运行于机器上的服务竞争 CPU 和其他资源。在高容量负载平衡上不推荐这种方法，但是在许多情况下不错。

软件负载平衡甚至可以将整个栈进一步下移到客户端本身。客户端负载平衡要求客户端知道哪些服务器可用，并进行自己的健康检查、服务器选择等。这通常用于内部服务，因为这些服务一般有更紧密的控制。客户端库可以从中心位置加载配置，对请求进行负载平衡，检测和绕过故障。缺点是，如果要更改算法或者修复缺陷，必须更改客户端库，这需要在任何地方进行更新。

6.6.4 机架

机架本身通常不会出现故障，它们只是一堆钢材，没有任何活动组件。但是，许多故障是遍及整个机架的。例如，机架可能只有一路电源供应或者网络上联链路，供机架上的所有设备共用。侵入式维护往往一次涉及一个机架。

因此，机架通常是一个故障域。实际上，对于大部分分布式系统，有意地将每个机架设计为单独的故障域，是比较容易管理的规模。

机架多样性

可以选择将一个服务分解到多个副本，并在每个机架上放置一个副本。这种安排下，服务就具备**机架多样性**（Rack Diversity）。DNS 服务就是一个简单的例子，每个 DNS 放在不同机架上，这样机架范围的故障不会造成服务中断。

在 Hadoop 群集中，为了保证安全，数据文件保存在多台机器上。确保任何数据块的至少一个副本在与其他数据块不同的机架上，该系统就可以实现机架多样性。

机架局部性

使服务组件在单个机架上保持完备性也能提供某些好处。在一个机架内部带宽很充足，

但是在机架之间的带宽较小。机架上的所有机器连接到顶部的同一个交换机。这个交换机有足够的内部带宽，任何机器可以与机架上的任何机器以全带宽通信，所有机器都可以同时达到这一点，这种方案称作无阻塞带宽（Non-blocking Bandwidth）。在机架之间，带宽就不那么充足了，机架上联链路往往 10 倍于机器之间的链接，但是它们是机架中所有机器（通常为 20 或者 40 台）使用的共享资源。机架之间的带宽存在争用。《A Guided Tour through Data-center Networking》（数据中心联网导览，Abts & Felderman, 2012）这篇文章以 Google 网络为例深入研究了主题。

因为机架内部的带宽很丰富且机架是一个故障域，服务组件往往设计为容纳在一个机架中。小的查询进入机架，使用许多带宽生成应答，然后发送小到中等大小的响应。这种模式很适合于带宽有限的情况。

然后，服务组件被复制到许多机架上。每个副本都具备机架局部性（Rack Locality）——在机架内保持完备。这种设计利用了高带宽和机架规模的故障域。

机架规模的副本有时候称作豆荚（Pod）。豆荚是完备的，往往组成自己的安全域。例如，计费系统可能由多个豆荚组成，每个豆荚都是完备的，设计用于处理特定客户组的计费。

Clos 网络

希望最终在公开市场上出现为整个数据中心内的任意两台机器提供无阻塞、全速连接的网络产品，是很合理的。我们从 1953 年（Clos 1953）开始就已经知道如何做到这一点了。当这种产品开始推出，将改变我们的服务设计方法。

6.6.5 数据中心

数据中心也可能是故障域。整个数据中心可能因为自然灾害、散热故障、电源故障或者挖掘机作业时切断所有网络连接而停止运行。

与机架多样性和机架局部性类似，也可以和机架内部一样实现数据中心多样性和数据中心局部性。数据中心内部的带宽通常很大，但是不像机架内部的通信那么快。数据中心之间的带宽通常较小，而且和数据中心内部的数据传输不同，它们往往是按照 GB 计费的。每个服务副本应该在数据中心内保持完备性，而整个服务应该具备数据中心多样性。Google 要求对面向用户的服务至少采用 $N+2$ 多样性。这样，当一个数据中心为维护而停机时，另一个数据中心因为不可预见的情况而停机不会影响到服务。

6.7 超载故障

分布式系统在面对因为临时流量浪涌、蓄意攻击或者自动化系统高速查询系统（可能是恶意的）而产生的高负载水平时必须具备弹性。

6.7.1 流量浪涌

系统对于临时的高负载应该具备弹性。例如，小型服务可能在被流行网站或者新闻广播提及之后超载。即使大规模的服务也可能因为某个副本出现故障时负载转移到其余副本而超载。

在面向用户的服务中处理这一问题的主要策略是优雅降级。这一主题在 2.1.10 节中已经介绍过。

动态资源分配

另一种策略是动态增加容量。利用这种方法，一个系统将检测服务超载，并从正在运行但未配置的闲置机器池中分配一台未使用的机器。一个自动化系统将配置该机器并用它为超载服务增加容量，从而解决问题。

拥有闲置容量的代价可能很高，但是这一代价可以通过使用共享池（Shared Pool）减轻。也就是说，由一个闲置机器池服务于一组服务。第一个超载的服务会分配这些机器，如果共享池足够大，就可以承受有不止一个服务同时超载的情况。在需求消失时，应该有一种机制，使服务交还分配的机器。

附加容量也可以在其他服务提供商那里找到。公共云计算提供商可以充当共享池。通常不需要为未用容量付费。

共享资源池不只适合于机器，也可以用于存储或者其他资源。

减载

另一种策略是减载（load shedding）。利用这种策略，服务拒绝一些用户，使其他用户能够得到好的体验。

做个类比，超载的电话系统不会突然断开所有现存通话，而是用一个“快速忙音”应答新的通话尝试，让这个人在以后重新尝试拨号。超载的网站也类似，给某些用户及时的应答，如简单的“过会回来”网页，而不是让他们在几分钟的等待之后超时。

减载的一个变种是停止某些可推迟到以后的任务。例如，低优先级的数据库更新将排队等待以后处理；存储用户声誉分数的社会化网络可以保存分数已经获得的事实而不加以处理；如果网络超载，可以延迟夜间大文件传输。

话虽如此，如果可延迟两个小时的任務被永久拖延，可能会造成问题。毕竟，它们的存在是有理由的。对于任何因为减载而被延迟的活动，必须有处理延迟的计划。建立服务水平协议（Service Level Agreement, SLA），以确定某些服务可以延迟的时长，识别应该进行的操作时限，缓解问题或者延长期限。低优先级更新可能在一定时间之后变成高优先级任务。如果许多系统因为减载而关闭，可以一次启动一个，让每个系统慢慢跟上。

为了管理这种情况，必须对系统有可见性，以便做出优先级决策。例如，知晓任务的寿命（它已经被延迟了多久），预测处理所需时间，并说明与最后限期的接近程度，运营人员就可以估计延迟项目应该继续的时间。

延迟的工作会降低质量

Google 网站搜索的旧版本有两个部分：面向用户的 Web 前端和接受及处理更新搜索索引（语料库）的系统。这些更新以大批次的形式到达，必须分布到每个前端。

搜索系统的质量按照所有 Web 前端上语料库的新鲜度作为计量。

监控仪表盘显示每个前端上分片的新鲜度。它列出了每个新鲜度“桶”中有多少个分片：最新、1 小时、2 小时、4 小时等。利用这种可见性，运营人员可以发现何时出现问题，增进对那些最过时的前端的理解。

如果系统超载，更新程序系统将被暂停，为处理更多负载留出资源。仪表盘使运营人员能够了解暂停的效果。他们可以不暂停高优先级更新，以维护最低的新鲜度。

6.7.2 DoS 和 DDoS 攻击

拒绝服务（Denial-of-service, DoS）攻击是通过发送大量查询使服务下线的企图。分布式拒绝服务（Distributed Denial-of-service, DDoS）攻击发生在互联网各处的许多机器被用来以协调一致的方式进行极大规模 DoS 攻击的时候。DDoS 攻击常常从僵尸网络（Botnet）启动，僵尸网络是全世界的大量计算机，它们已经被成功渗透并遭到集中控制，而其所有者对此一无所知。

拦截这些请求通常不能成功地防御 DDoS 攻击。攻击者可能伪造数据包，隐瞒攻击来源，从而使你无法构造过滤器，在不拦截合法流量的情况下阻止进攻。如果它们来自于固定来源，不响应这些请求仍然会因为接受攻击而损失带宽——仅仅这样就可能使网络超载。这种攻击必须从网络之外拦截，通常要依靠所连接的 ISP。大部分 ISP 不提供此类过滤。

最佳的防御措施是拥有比攻击者更大的带宽。大部分 DDoS 攻击都涉及数千台机器，因此这一点难以想象。超大型公司可以使用这样的防线。较小的公司可以使用 DDoS 攻击缓解服务。许多 CDN 供应商（参见 5.8 节）提供这种服务，因为它们有足够的带宽。

有时候，DDoS 攻击的目标不是耗尽带宽，而是消耗大量处理时间或者负载。例如，人们可能发现一个小的查询需要大量资源来响应。在这种情况下，可以用前面描述的禁止查询列表拦截此类查询，直到软件发行版本修复该问题。

6.7.3 抓取攻击

抓取（Scraping）攻击是一个自动化进程，像 Web 浏览器一样查询信息并从接收到的 HTML 页面中提取（抓取）有用信息。例如，如果你想要以往发行的每本书籍的列表，但是不想向图书馆供应公司付费取得数据库，可以编写一个程序，向 Amazon.com 发送数百万个搜索请求，解析 HTML 页面，提取书名以构建自己的数据库。对 Amazon 的这种使用可以视为一次攻击，因为这违反了该公司的服务条款。

必须防御这种攻击，避免信息盗窃、避免某人违反服务条款，因为很快的抓取程序与

DoS 攻击等价, 这种攻击的检测通常通过所有前端向中央抓取检测服务报告所接收查询的相关信息完成。

抓取检测程序就任何可疑的攻击警告前端。如果特定来源涉及攻击的可信度很高, 前端可以拦截或者拒绝应答查询。如果攻击源可信度低, 前端可以其他方式响应。例如, 它们可能使用验证码或者其他区分人类和机器输入的系统, 要求用户证明自己是个人。

有些抓取是允许, 甚至是必要的。抓取检测程序应该有一个白名单, 允许搜索引擎爬虫和其他允许的代理完成其工作。

6.8 人为错误

在我们设计了对硬件及软件故障更有弹性的系统时, 剩下的故障很可能是由于人为错误。虽然这看上去显而易见, 但是直到 2003 年发表的具有开创性的论文《Why Do Internet Services Fail, and What Can Be Done about It?》(“为什么互联网服务会失效, 对此可以采取什么措施?”, Oppenheimer、Ganapathi & Patterson, 2003), 这一趋势才得到认可。

处理人为错误的策略可以分为获得更好的人、从循环中消除人为因素以及检测并绕过人为错误。

我们通过采用更好的运营实践获得更好的人, 特别是演练最需要改进的技能和行为的人(参见第 15 章)。

通过自动化可以在循环中消除人为因素, 人可能疏忽, 在规程中没有对错误进行很多检查, 但是自动化程序一旦写成, 总是会检查其工作(参见第 12 章)。

检测和绕过人为错误也是自动化的功能之一。**预先检查**(pre-check)是检查输入, 并在测试失败时阻止进程运行的一种自动化手段。例如, 预先检查可以验证最近编辑的配置文件没有语法错误, 且符合某种其他质量标准。无法通过预先检查, 则配置文件不能投入使用。

虽然预先检查的意图是避免问题的发生, 现实是它们往往落在经验之后。也就是说, 在每次运行中断之后, 我们添加新的预先检查, 避免相同的人为错误造成未来的运行中断。

另一种常见的预先检查是检查大的更改。如果典型的更改通常只由几行代码组成, 预先检查可能要求大于特定数量代码行的更改经过附加的批准。这种更改可能发生在输入的大小、当前输入和新输入之间更改的行数或者当前输出和新输出之间更改的行数。例如, 某个配置文件可能用于控制生成其他文件的系统, 输出的增长超出某一比例可能触发附加的批准。

对人为错误保持弹性的另一种手段是由两个人检查所有更改。许多源代码控制系统可以配置为在第二个用户批准之前不接受来自某个用户的更改。这样, 所有通过源代码存储库中文件更改进行的系统管理工作都受到第二双眼睛的检查。这是 Google 非常常见的运营方法。

6.9 小结

弹性是系统建设性地处理故障的能力，弹性系统检测并绕过故障。

故障是运营的正常组成部分，可能发生在任何级别上。大型系统会放大小故障的风险。如果机器足够多，概率为百万分之一的故障可能每天都发生。

故障有许多来源。软件可能因为缺陷而意外失效，也可能因为有意地阻止更糟糕的情况发生而中断。硬件也可能出现故障，其范围从最小的组件直到最大型的网络。故障域可以为任意大小：设备、计算机、机架、数据中心甚至整个公司。

系统中的容量为 $N+M$ ，其中 N 是用于提供服务的容量， M 是可用的备用容量，用于防范故障。 $N+1$ 容错系统可以在一个单元故障中存活下来，保持正常运行。

绕过故障的最常见方法是通过服务的复制。服务可以在每个故障域复制一次或者多次，提供比该域更大的弹性。

故障也可能来自于导致系统超载的外部来源和人为错误。对于几乎所有可以想象的故障都有对策。我们无法预测所有故障，但是可以规划，设计解决方案，排定实施优先级并重复上述过程。

练习

1. 分布式计算系统中最主要的故障源是什么？
2. 软件、硬件还是人为故障中哪种故障最常见？证明你的答案。
3. 选择一种弹性技术，并举出一个故障的例子及弹性技术避免用户可见运行中断的方法。
对 6.5 节、6.6 节、6.7 节和 6.8 节中的每种技术重复上述活动。
4. 如果使用负载平衡器，系统自动成为可伸缩和具备弹性的。你同不同意这一观点？证明你的答案。
5. 你的环境中使用何种弹性技术？
6. 你希望在当前环境中的哪个位置增加弹性？描述你将要做的更改和应用的技术。
7. 在你的环境中，举出负载下优雅降级的一个例子，如果不存在，解释你将如何实施它？
8. RAID5 阵列可以达到多大？例如，达到多大时奇偶校验方案可能遗漏一个错误？在 MTBF 使系统处于第二次故障风险中之前，重建时间为多长？
9. 本章中提到了一个短句：“适当饮食、锻炼，人还是会死。”解释这句话与分布式计算的关联。

机群：虚拟或者物理机器。

错误（随时待命）：成为运行中断或者警告的第一响应者。

服务器：提供功能或者 API 的软件（不是硬件部件）。

服务器组成的用户可见系统或者产品。

在还没有公开通告的情况下启动新服务。

部门提供一定缓冲，在许多人发现之前修复问题或者更新系统。

第二部分 Part 2

运营：运行系统

1. 分布式系统运营

要理解分布式系统运营，必须首先理解它与系统开发之间的关系。运营是系统开发过程中一个至关重要的阶段，它与系统开发之间的关系，以及系统运营的基本技术。

7.1.1 SRE 运营模型

- 第 7 章 分布式世界中的运营
- 第 8 章 DevOps 文化
- 第 9 章 服务交付：构建阶段
- 第 10 章 服务交付：部署阶段
- 第 11 章 升级运行中的服务
- 第 12 章 自动化
- 第 13 章 设计文档
- 第 14 章 随时待命
- 第 15 章 灾难准备
- 第 16 章 监控基础知识
- 第 17 章 监控架构与实践
- 第 18 章 容量规划
- 第 19 章 建立 KPI
- 第 20 章 卓越运营

级的服务。IT 服务往往由外部承包商构建，一旦系统运行，运营团队就需要负责维护。

SR 运营维护团队不断发展，以处理更多的流量和更大的工作负载。团队必须得到处理的速度）和整体吞吐量都需要管理。效率成为一个关键点，因为团队需要快速响应（比如，当故障发生时，团队需要快速响应）。团队需要快速响应（比如，当故障发生时，团队需要快速响应）。

6.9 小结

故障是系统运行中不可避免的一部分，弹性系统检测并绕过故障。

故障是系统运行中不可避免的一部分，弹性系统检测并绕过故障。

故障是系统运行中不可避免的一部分，弹性系统检测并绕过故障。

故障是系统运行中不可避免的一部分，弹性系统检测并绕过故障。

如果机器足够多，故障是系统运行中不可避免的一部分，弹性系统检测并绕过故障。

Chapter 7

第7章

分布式世界中的运营

系统中的容量为 $N+M$ ，其中 N 是用于提供服务的容量， M 是可用容量。

范故障。 $N+1$ 容错系统可以在一个单元故障中存活下来，保持正常运行。

绕过故障的最常见方法是通过冗余，即复制一份或者多份。

提供比该域更大的弹性。

故障也可能来自于导致系统超载的外部来源和人为错误。对于几乎所有可以想象的故障都有对策。

我们无法预测所有故障，但是可以规划，设计解决方案，排定实施优先级。

重复上述过程。

组织学习的速度很快将成为竞争优势唯一的可持续来源。

——Peter Senge

练习

第一部分讨论了如何构建分布式系统。现在，我们讨论如何运行这些系统。

保持系统运行的工作称作**运营**（Operation）。更确切地说，运营是为保持系统以满足或者超过服务水平协议（Service Level Agreement, SLA）规定的运营参数的方式运行所做的工作。运营包括服务生命期的所有方面：从初次启动到最终退役，以及两者之间的所有工作。

运营工作的焦点往往是可用性、速度和性能、安全性、容量规划和软件/硬件升级。这些工作中的失败会造成系统不可靠。如果服务速度缓慢，用户会认为它已经被破坏。如果系统不安全，外部人员可能使其下线。没有合适的容量规划，服务将会超载并失效。升级如果操作不当，会造成停机时间。如果升级完全没有任何作用，缺陷就不能修复。因为所有这些活动最终会影响系统的可靠性，Google 将其运营团队称作网站可靠性工程（Site Reliability Engineering, SRE）。许多公司也照搬这一叫法。

运营是一项团队活动，不是由一个人完成的，而是由一个团队协作完成的。因此，我们所描述的大部分内容都是帮助你以团队的方式而非一组个人工作的过程和策略。在某些公司，过程似乎是拖慢节奏的官僚主义“迷宫”。正如我们在此描述的——更重要的是，在我们的专业经历中——好的过程使超大规模计算系统的运行成为可能。换言之，过程使团队一次又一次地做正确的事情成为可能。

必知术语

创新：做我们之前未做过的（好）事。

机器：虚拟或者物理机器。

值班（随时待命）：成为运行中断或者警告的第一响应者。

服务器：提供功能或者 API 的软件（不是硬件部件）。

服务：有一个或者多个服务器组成的用户可见系统或者产品。

软启动（Soft Launch）：在没有公开通告的情况下启动新服务。这样，流量随着口碑传播而慢慢增长，为运营部门提供一些缓冲，在许多人发现之前修复问题或者扩展系统。

SRE：网站可靠性工程师，Google 用这一术语表示维护线上服务的系统管理员。

利益相关方：在项目的成功上有利益关系的人或者组织。

本章从一些运营管理的背景开始，然后讨论运营服务生命期，最后讨论典型的运营工作策略。所有这些主题将在后续章节中展开讲解。

7.1 分布式系统运营

要理解分布式系统运营，必须首先理解它与典型企业 IT 之间的不同，还必须理解运营和开发人员之间紧张关系的根源，以及伸缩性操作的基本技术。

7.1.1 SRE 和传统企业 IT 的对比

系统管理是一个连续体，一端是典型的 IT 部门，负责传统桌面和客户 - 服务器计算基础设施，往往被称作企业 IT。在另一端的是 SRE（或者类似的术语），负责分布式计算机环境，往往与网站和其他服务相关。虽然这只是一个概括，但是能够说明某些重要的差异。

SRE 不同于企业 IT 部门，因为 SRE 倾向于将提供一个服务或者一组具有良好定义的服务作为焦点。传统的企业 IT 部门倾向于在桌面服务、后台服务和两者之间的一切（“有电源插头的任何东西”）上有广泛的职责。SRE 的客户通常是服务的产品管理人员，而 IT 的客户是最终用户本身。这意味着，SRE 的工作集中在几个精选的业务指标，而不是被用户吸引到许多个方向上，每个 SRE 各有自己的优先事务。

另一个差别是对正常运行时间的态度。SRE 维护有 24×7 运行要求的服务，因此他们的焦点是避免问题和执行复杂但非侵入性的维护规程，而不是对运行中断做出反应。IT 通常在停机时间的安排上有一定的灵活性，并且具有专注于服务在运行中断时如何快速恢复的 SLA。从 SRE 的角度，停机时间是必须避免的，即使处于维护中，服务也不应该停止。

SRE 倾向于管理因为新软件发行和功能增加而不断变化的服务。IT 倾向于运行较少升级的服务。IT 服务往往由外部承包商构建，一旦系统稳定，这些人就会离开。

SRE 所维护的系统不断扩展，以处理更多的流量和更大的工作负载。延迟（特定请求得到处理的速度）和整体吞吐量都需要管理。效率成为一个关注点，因为每台机器浪费少量时间，在成百上千台机器上就会造成巨大的浪费。在 IT 中，系统往往是为每年中预期的工

作负载增加较为适度的环境所构建的。在这种情况下，切实可行的战略是构建足够大的系统，以处理接下来几年中计划的工作负载，此后系统应该会被替换。

因为这些需求，SRE 倾向于定制系统，在自制或者从开源或其他第三方组件集成而来的平台上构建，它们不是“现成”或者整套承包的系统，必须主动地管理，而 IT 系统从初始交付状态起可能不做更改。由于这些差异，分布式计算服务最好由单独的团队管理，这些团队具备不同的管理层，采用定制的运营和管理方法。

虽然有许多差异，最近 IT 部门已经开始出现和 SRE 环境中类似的运行时间和可伸缩性需求。因此，分布式计算的管理技术很快会在企业中得到采用。

7.1.2 变化和稳定性的对比

对稳定性的渴望和对变化的渴望之间有难以调和的紧张关系。运营团队更喜欢稳定性；开发人员则渴望变化。考虑一下每个小组在年终绩效审核时的评估方法。开发人员因为编写能够投产的代码而受到表扬。对服务造成明显差异的变化所得到的奖励超过其他任何成就。因此，开发人员希望常常投产新的发行版本。相反，运营部门因为实现 SLA 的相容性而得到奖励，其中大部分与正常运行时间相关，因此，稳定性是头等大事。

系统始于一个稳定性基线，然后发生变化。所有变化都有某种失稳效应。最终，系统会重新趋于稳定，这通常需要某种干预。以上过程被称作变化-不稳定循环（Change-instability Cycle）。

所有的软件试运行都会影响稳定性。更改可能引入缺陷，需要通过变通方法和新软件发行版本修复。没有引入新缺陷的发行仍然会造成失稳效应，因为必须处理从将要升级的机器中切换出来的工作负载。非软件更改也有失稳效应。网络的变化传播到整个网络时可能使局域网稳定性下降。

因为运营对稳定性的渴望和开发人员对变化的渴望之间的冲突，必须有保持平衡的机制。策略之一是改善稳定性的工作优先于增加新功能的工作。例如，缺陷修复的优先级应该高于功能请求。采用这种方法，主发行版本引入许多新功能，接下来的几次发行专注于缺陷修复，然后新的主发行版本再次启动这一循环。如果工程管理将焦点放在新功能上而忽略缺陷修复，结果就是系统会越来越不稳定，直到失去控制。

第二种策略是协调开发人员和运营人员的目标。双方都对 SLA 依从性和系统速度（变化率）负有责任。两者的年度审核中都有与 SLA 依从性相关的部分，也都有和及时交付新功能相关的部分。

成功协调双方目标的组织重新构造自身，使开发人员和运营人员像一个团队一样工作。这是 DevOps 活动的前提，在第 8 章中将进行描述。

另一种策略是为稳定性改善和新功能预算时间。软件工程组织通常有估算软件请求规模或者预计完成时间的手段。每次新发行都有某种规模和时间预算，在预算内分配一定数量的稳定性改善工作。2.2.2 节最后的案例研究是这种方法的一个例子。类似地，这种分配

可以通过为稳定性相关代码更改而分配的专门人员来实现。

预算也可以基于 SLA。每月预期一定量的不稳定情况，这可以看作一种预算，每次试运行使用某些预算，与不稳定相关的缺陷也是如此。开发人员可以通过致力于改进导致不稳定的代码，最大化每个月所能进行的试运行数量。这就可以造成正反馈循环。Google 的错误预算 (Error Budgets) 就是一个例子，19.4 节将进行全面的解释。

7.1.3 SRE 定义

在公开列举之前，Google 在 10 多年的时间内对 SRE 核心实践进行了提炼。在第一次 USENIX SREcon 上的专题演讲中，Google 网站可靠性工程副总裁 Benjamin Treynor Sloss (2014) 列举了这些做法：

网站可靠性实践

- 1) 只雇用编码人员。
- 2) 为服务制定一个 SLA。
- 3) 根据 SLA 计量和报告性能。
- 4) 使用错误预算和关卡式投放 (Gate Launch)。
- 5) 为 SRE 和开发人员设置共同的人员池。
- 6) 过度的运营工作“溢出”到开发团队。
- 7) 将 SRE 运营负荷限制在 50%。
- 8) 开发团队分担 5% 的运营工作。
- 9) 值班团队应该在一个位置至少有 8 个人，或者在多个位置上每个位置有 6 个人。
- 10) 以每个值班轮次最多两个事件为标准。
- 11) 对每个事件进行事后剖析。
- 12) 事后剖析不是为了指责某人，而是专注于过程和技术而非人。

网站可靠性工程的第一条原则是 SRE 必须会编码。SRE 可能不是全职的软件开发人员，但是他应该能够编写代码解决复杂的问题。当被要求重复执行某项任务 30 次时，SRE 应该做完前两次就感到厌烦，自动化其余的任务。SRE 必须有足够的软件开发经验，可以和开发人员在同等的水平上沟通，能够鉴别开发人员所做的工作，以及计算机所能做和不能做的事情。

当 SRE 和开发人员来自共同的人员池时，意味着该项目分配了一定数量的工程师。这些工程师可能是开发人员或者 SRE。最终结果是，团队中每增加一位 SRE 就意味着少了一位开发人员。与此相反，大部分公司中系统管理员和开发人员从具有单独预算的团队中分配。项目希望最大化开发人员的数量很合理，因为他们要编写新的功能。共同的人员池鼓励开发人员创建可以高效运营的系统，最小化所需的 SRE 数量。

鼓励开发人员编写最小化运营负担的代码的另一种手段是要求过多的运营工作“溢出”到开发人员。这种方法阻止开发人员采取捷径，造成过分的运营负担。开发人员应该分担

这种压力，同样，要求开发人员执行 5% 的运营工作，开发人员就能够和运营的现实保持一致。

在 SRE 团队中，将运营负载控制在 50% 限制了手工劳动的数量。例如，手工劳动的投资回报低于编写代码替代这些劳动。这将在 12.4.2 节讨论。

许多 SRE 实践和寻求对变化的渴望与对稳定性的渴望之间的平衡相关。最重要的是 Google 的 SRE 实践——错误预算，这将在 19.4 节详细解释。

错误预算的核心是 SLA。所有服务都应该有 SLA，规定系统应该达到的可靠性。SLA 成为最终计量所有工作的标准，第 16 章将讨论 SLA。

运行中断或者其他重大 SLA 相关事件之后，应该创建书面的事后剖析，包含所发生情况的细节，以及未来如何避免这种状况的分析和建议。这种报告在公司内共享，以便整个组织能够吸取经验。事后剖析专注于过程和技术，而不是找出责任人。14.3.2 节将介绍事后剖析。值班人员负责对任何 SLA 相关事件做出响应，并制作事后分析报告。

值班不仅是响应问题的一种手段，也是减少未来问题的手段，必须以不对值班人员造成莫名压力的方式进行，并且推动鼓励长期修复和问题预防的行为。值班团队由同一位置的至少 8 位成员，或者两个位置的各 6 位成员组成，这种规模的值班团队往往足以保证其技能不会过时，他们的轮班时间应该足够短，使得每个班次所遇到的运行中断事件不会超出两个。因此，每个成员有足够的时间跟踪每个事件，执行必要的长期解决方案。值班管理的这种方式是第 14 章的主题。

其他公司已经采用了 SRE 作为管理线上生产服务的系统管理员的头衔。每个公司对这一角色采用不同的做法。Google 中定义 SRE 的实践是成功的核心。

7.1.4 大规模运营

分布式计算中的运营是规模化的。分布式计算涉及数百台（甚至往往达到数千台）共同工作的计算机，因此，运营不同于传统的计算管理。

人工过程没有伸缩性。当任务是人工进行的时，如果任务增加一倍，就需要两倍的人力。因此，如果过程涉及人工操作，就无法扩展到数千台机器、服务器或者进程。相反，自动化具备伸缩性。代码编写一次，可以使用数千次。涉及许多机器、进程、服务器或者服务的过程应该自动化。这一思路适用于机器分配、操作系统配置、软件安装和故障监控。自动化不是“锦上添花”，而是“雪中送炭”。（自动化是第 12 章的主题）

运营自动化时，系统管理更像一个生产线而不是一项“手艺”。系统管理员的工作从工人变成维护生产线机器人的技师。大规模制造技术已经切实可行，我们可以从制造中借鉴运营方法。例如，通过从每个生产阶段收集指标，可以应用统计分析，帮助我们改善系统吞吐率。持续改进（Continuous Improvement）等制造技术是 DevOps 三条道路的基础（参见 8.2 节）。

没有自动化的事务有 3 类：应该自动化而尚未自动化的、不值得自动化的以及无法自

动化的人工过程。

尚未自动化的任务

创建、测试和部署自动化需要时间，所以总是有一些事务等待自动化。我们总是没有足够的时间自动化所有事务，所以必须排定优先级，明智地选择方法（参见 2.2.2 节和 12.1.1 节）。

对于没有自动化或者尚未自动化的过程，创建规程化文档（称作**剧本**），使过程变为可重复、一致性的。好的剧本使过程在未来更易于自动化。自动化中最为困难的部分往往是精确描述过程。如果剧本做到这一点，实际的编码相对简单。

不值得自动化的任务

某些任务不值得自动化是因为它们不经常发生、自动化难度太大或者过程太经常变化而无法自动化。自动化需要投入时间和精力，并不总有合理的投资回报（ROI）。

不过，有一些值得自动化的常见案例。在自动化这些案例时，往往可以合并或者消除较少见的情况（**边缘情况**）。在许多情况下，新自动化的常见案例提供超卓的服务，以致处于边缘情况的客户突然失去了其独特需求。

自动化常见案例的好处

某家公司中有 3 种配给虚拟机的方法。这三种方法都是人工过程，客户往往需要等待几天，系统管理员才有空完成该任务。自动化配给项目因为处理 3 种不同情况的复杂性而止步不前。两个较不常见案例的用户们要求配给过程不同，是因为这些虚拟机（在他们自己眼中）是独特而美丽的“雪花”。他们提出非常严肃的（经验）证据，并且猛烈地挥动双手证明自己的观点。为了推进项目，公司决定只自动化最常见的案例，并承诺将在以后加入两种边缘情况。

比起最初华而不实的配给系统，这样的系统实施起来容易得多。利用初始自动化，配给时间缩短到几分钟，可以在没有系统管理员干预的情况下进行。配给甚至可以在夜间和周末进行。此时发生了一件令人吃惊的事情。其他两种情况的用户突然发现他们的独特性消失了！他们采用了这种自动化方法，系统管理员从未自动化两种边缘案例，配给系统保持较低的复杂性、易于维护。

无法自动化的任务

有些任务无法自动化，因为它们是人工过程：维护和利益相关方的关系、管理大型采购项目的招标过程、评估新技术或者在团队内部协商值班表的安排。虽然通过自动化无法消除这些任务，但是可以合理化：

- 与利益相关方的许多互动可以通过更好的文档消除。利益相关方如果得到介绍性文档、用户文档、最佳实践建议、样式指南等，就可能更容易自给自足。如果服务供许多其他服务或者服务团队使用，有很好的文档就更为重要。视频指南也很有用，

如果只是制作记录自己演示的视频，需要的工作量也不大。

- ❑ 和利益相关方的一些互动可以通过提供自助式的常见请求消除。不需要单独会见以理解客户的容量需求，他们的预测可以通过 Web 用户界面或者 API 收集。例如，如果你为数百个其他团队提供服务，预测可以成为项目经理的全职工作。另外，利用合适的自动化措施，和公司的供应链管理系统集成也是轻而易举的。
- ❑ 新技术评估可能是劳力密集型的工作，但是如果识别了常见案例，端到端的过程可以转变为流水线式过程并加以优化。例如，如果要购买数以千计的硬盘，定期混合且全面评估之后才增加一种新型号是明智的做法。评估过程应该标准化和自动化，结果自动存储以供分析。
- ❑ 自动化可以替代或者加速团队过程。建立值班安排可能因为团队成员争夺重要节日的休假权而造成混乱。自动化措施可以将此过程变成自助系统，允许人们列出空闲时间，制作出下几个月的最优安排。因此，它能够更好地解决问题，减少紧张情绪。
- ❑ 通信、状态和过程跟踪等元过程可以通过在线系统加以促进。随着团队的成长，仅仅跟踪各方之间的互动和沟通就可能成为负担。自动化可以消除每个人的许多手工劳动。例如，让人们在整个批准过程中看到订单状态的 Web 系统能够消除状态报告的需求，人们可以只处理异常和问题。如果过程在团队之间有许多复杂的移交工作，提供状态仪表盘、在移交发生时自动通知团队的系统可以减少项目经理的人数。
- ❑ 最佳的过程优化是消除。被消除的任务不需要执行或者维护，也不会有缺陷或者安全漏洞。例如，如果生产机器运行 3 种不同的操作系统，将这个数字缩减为两个会消除许多工作。如果为其他服务团队提供一项服务，每个新团队需要一个冗长的批准过程，自动批准某些类型的用户以合理化批准过程可能更好。

7.2 服务生命周期

运营负责的是整个服务生命期（Service Life Cycle）：启动、维护（常规和紧急）、升级和退役。每个阶段都有独特的需求，所以你需要以不同方式管理每个阶段的策略。

生命期的各阶段如下：

- ❑ **服务启动**：第一次启动服务。服务诞生，初始客户使用，发现并补救投放前未发现的问题（7.2.1 节）。
- ❑ **紧急任务**：处理异常或者意外事件。这包括处理运行中断，更重要的是检测和修复造成运行中断的条件（第 14 章）。
- ❑ **非紧急任务**：执行所有正常运作系统所需要的人工任务。这可能包括定期（每周或者每月）维护任务（例如，为每月计费活动作准备）和处理来自用户的请求（例如，

另一个内部服务或者团队请求启用服务)(7.3节)。

- **升级**：部署新软件发行版本和硬件平台。这一点做的越好，公司就能越积极地尝试新事物和创新。每个新软件发行版本在部署之前构建和测试，测试包括开发人员进行的系统测试和运营部门进行的用户验收测试(UAT)。UAT可能包括验证没有性能退化(Performance Regression)(性能的意外降低)的测试。漏洞评估用于检测安全问题，新硬件必须经过硬件认证(Hardware Qualification)，以测试兼容性、性能退化和运营过程中的任何更改(10.2节)。
- **退役**：关闭服务，是服务启动的反面：删除剩余的用户，关闭服务，删除任何相关服务配置中对该服务的引用，交还任何资源，存档旧数据，并在硬件更改用途、出售或者处置之前擦除数据(7.2.2节)。
- **项目工作**：执行需要分配专门资源和规划的大型任务。虽然不是服务生命期的直接组成部分，在此期间可能发生比其他任务更大规模的工作。这方面的例子包括修复重复、间歇出现的故障，和利益相关方一同研究路线图并规划产品的未来，将服务转移到新的数据中心，以新方式扩展服务(7.3节)。

这里列出的大部分生命周期阶段都将在本书的其他地方详细介绍。服务启动和退役将在下面详细介绍。

7.2.1 服务启动

最令人尴尬的事莫过于新服务公开启动失败。我们常常看到某个新服务成功启动，但是接收的流量过大而超载，并停止运行。这很有讽刺意味，但并不好笑。

每当我们启动一个新服务，就会学到某些新东西。如果很少启动新服务，在下一次启动时想起这些教训就很难。因此，如果启动次数很少，我们应该维持一个检查列表，并记录下次应该记得完成的工作。随着检查列表在每次启动中增长，我们越来越擅长于启动服务。

如果经常启动新服务，可能有许多人参加启动。有些人的经验可能较少，在这种情况下，我们应该维护一个检查列表，以分享自己的经验。检查列表项目的每次增加都会增进我们的组织记忆(Organizational Memory)——组织内部的知识集合，从而使组织变得更有智慧。

常见的一个问题是，其他团队可能没有意识到启动的规划需要花费精力。他们可能没有为此分配时间，而在接近启动日期时才给了运营团队意外的一击。这些团队不知道检查列表所要预防的潜在陷阱和问题。因此，启动检查列表应该在文档中频繁提到，在产品经理之间传达，并使之容易访问。当和运营团队接洽，希望启动某项服务的团队在整个开发期间都以检查列表作为指南，那就是最佳的情况。这样的团队已经“做了功课”，他们已经在开发产品期间并行研究了检查列表中的项目。这不是偶然发生的。检查列表必须可用、广而告之并且成为公司文化的一部分。

一种简单的策略是创建一个必须在启动之前完成操作的检查列表。更复杂的策略是在检查列表中包含一系列可由启动准备工程师（LRE）或者启动委员会审核的问题。

下面是启动就绪审核检查列表的一个样板：

启动就绪审核问卷

本文档的目的是收集信息，供启动准备工程师（LRE）在批准新服务启动时评估。请在与 LRE 会面之前完成该问卷。

☐ 启动总体信息：

- ☐ 服务名称
- ☐ 启动日期 / 时间
- ☐ 是软启动还是硬启动

☐ 架构：

- ☐ 描述系统架构。如果可能，链接到架构文档。
- ☐ 在单机、机架和数据中心故障时如何进行故障切换？
- ☐ 系统设计如何在常规条件下伸缩？

☐ 容量：

- ☐ 预期的初始用户容量和 QPS。
- ☐ 如何达到这一数字？（链接到负载测试和报告）
- ☐ 如果初始容量 2 倍于预期容量，预计会发生什么情况？5 倍时呢？（链接到紧急容量文档）
- ☐ 预计的外部（互联网）带宽使用
- ☐ 在 1、3 和 12 个月之后的网络和存储需求（链接到来自网络和存储团队容量规划人员的确认文档）

☐ 依赖性：

- ☐ 本服务依赖于哪些系统？（链接到依赖性 / 数据流图）
- ☐ 这些依赖性有哪些 RPC 限制？（链接到限制和可以处理流量的外部团队的确认）
- ☐ 如果超过这些 RPC 限制，会发生什么？
- ☐ 对于每个依赖性，列出请求和明确认可新服务使用这一依赖性（以及 QPS）的单据编号。

☐ 监控：

- ☐ 所有子系统是否都在监控之下？描述监控策略并记录所监控的系统。
- ☐ 所有重要子系统是否都有仪表盘？
- ☐ 是否有指标仪表盘？它们是否采用业务而非技术性术语？
- ☐ 上月的“假警报”次数是否低于 x？
- ☐ 在典型的一周内接收到的警报数量是否小于 x？

☐ 文档：

○ 是否存在一个剧本，包含所有运营任务和警报条目？

○ 由 LRE 审核每个条目的准确性和完整性。

○ 未消除的相关文档缺陷数量是否小于 x ？

□ 值班：

○ 是否完成了下 n 个月的值班安排？

○ 值班安排是否使每个班次可能得到的警报少于 x 次？

□ 灾难准备：

○ 如果第一天的使用率达到预期的 10 倍，有何计划？

○ 备份是否有效，是否进行过回复测试？

□ 运营健康：

○ “垃圾警报”是否及时调整或者更正？

○ 是否将缺陷记入文档，以升级问题的可见性，即使是人所共知解决方案的小烦恼或者问题？

○ 稳定性相关缺陷的解决是否优先于新功能？

○ 是否部署系统，确保未消除缺陷数量较低？

□ 批准：

○ 市场部是否已经批准了所有标志、措辞方式和 URL 格式？

○ 安全团队是否已经审核和批准服务？

○ 是否完成了隐私审计，并解决了所有问题？

因为启动很复杂，有许多的活动部件，我们建议由一个人（启动主管，launch lead）担任领导或者协调人的角色。如果开发人员和运营团队相互独立，可以每方选择一人作为代表。

启动主管通过检查列表开展工作，委派工作，记录任何遗漏的缺陷并跟踪所有问题，直到启动得到批准和执行。启动主管还可能负责协调启动后问题的解决。

案例研究：Google 自助启动

Google 启动许多服务，因此需要某种合理化启动过程，以及由某个团队独立发起启动活动的方法。除了为技术人员提供 API 和门户之外，启动就绪评估（Launch Readiness Review, LRR）使启动过程变成自助服务。

LRR 包含一个检查列表和实现每个项目的指南。指派一名 SRE 工程师指导团队完成该过程，确保达到很高的标准。

有些检查列表项目是技术性的——例如，确保 Google 负载平衡系统正常使用。另一些项目是警告性的，是为了避免启动团队重复其他团队过去所犯的错误。例如，某个团队因为所接受的用户数 10 倍于预期而导致启动失败。没有关于如何处理这种情况的计划。LRR 检查列表要求团队创建一个处理这种情况的计划，并说明它已经预先经过测试。

其他检查列表项目与业务相关，要求市场、法律和其他部门同意启动。每个部门有自己的检查列表，SRE 团队只在验证所有签收完成之后才允许服务可见于外部。

7.2.2 服务退役

退役 (Decommissioning 或简写为 decomm) 或者关闭服务包含 3 个主要阶段：删除用户、资源分配解除和资源清理。

删除用户往往是产品管理任务，通常包括使用户知道自己必须转移。有时候，将客户转移到另一个服务是技术性问题。用户数据可能需要移动或者存档。

资源分配解除可能涵盖许多方面。可能需要删除 DNS 条目、关闭机器电源、禁用数据库连接等等，通常涉及复杂的相互依赖。在最后一个用户离开服务之前，往往什么事也做不了。某些资源无法在其他资源之前解除分配，诸如此类。例如，在机器不再使用之前，DNS 条目通常不能删除。如果其他服务的分配解除依赖于网络连接性，网络连接必须保留。

资源清理包括安全擦除磁盘和其他媒体，以及所有硬件的处置。这些硬件可以改变用途、出售或者废弃。

如果退役没有正确进行，或者遗漏了项目，资源将保持分配状态。随时间推移增加的检查列表有助于确保退役全部完成、任务以合适的顺序执行。

7.3 运营团队组织策略

运营团队必须完成工作，因此团队需要某种策略，确保接受、安排和完成所有工作。一般来说，运营工作有 3 个来源，这些工作项目分为 3 类。为了理解组织团队的最佳方式，首先必须理解这些来源和分类。

这 3 种工作来源是生命期管理、与利益相关方的互动和过程改进及自动化。生命期管理是涉及服务运行的运营工作。与利益相关方的互动指的是维护使用及依赖服务的人们之间的关系，对其请求排定优先级并实施。过程改进和自动化是由企业对持续改进的渴望所激发的工作。

不管来源为何，工作都可以归入 3 个宽泛的类别中的一种：

- ❑ **紧急问题：**运行中断，表明可能出现运行中断的可预防问题以及其他团队的紧急请求。通常是监控系统通过 SMS 或者传呼机发出警报所触发的（第 14 章）。
- ❑ **常规请求：**过程性工作（尚未自动化的可重复过程）、非紧急问题报告、信息性问题和更大项目的初始咨询。通常由请求单据系统发起（14.1.3 节）。
- ❑ **项目工作：**演化系统的小型 and 大型项目。按照团队选择的项目管理风格进行管理（12.4.2 节）。

为确保所有工作来源和分类吸引注意力，我们推荐如下简单组织原则：除了确保紧急

问题立刻引起注意，并对非项目客户请求进行筛选并及时跟进以外，团队成员应该始终致力于项目工作。

更确切地说，在任何时刻，一位团队成员的最高优先任务应该是响应紧急情况，另一位团队成员的最高优先级任务应该是响应常规请求，团队的其余成员应该专注于项目工作。

这与运营团队经常采用的工作方式不同：每个人都从一个紧急情况奔向另一个紧急情况，没有时间从事项目工作。如果没有致力于改善这种情况，团队将不断地应付各种紧急情况，最终耗尽其精力。

重大的改善来自于项目工作。项目工作需要集中力，如果你不断被紧急的问题和请求所打断，就无法完成项目。如果整个团队都专注于紧急情况和请求，就没有人进行项目工作。

将运营团队组织为3个子团队，每个专注于一个工作来源或者一类工作，是很有吸引力的做法。但是这两种方法都会造成职责“竖井”。过程改善最好由涉及过程的人完成，而不是由旁观者完成。

为了实施我们推荐的策略，团队的所有成员将专注于其主要优先工作——项目工作。但是，团队成员在发生紧急问题时轮流负责。这一职责被称作**值班（Oncall）**。同样，团队成员轮流负责其他团队的常规请求，这一职责称作**单据任务（Ticket duty）**。

值班任务和单据任务常常安排为一个循环。例如，由8人组成的团队可能使用8周的循环，每个人值班一周：响应警报，将其余的时间花在项目上。每个人在不同周分配单据任务：首先专注于请求单据的筛选和响应，只在剩余时间致力于其他项目。这使团队成员可以在循环中的6周内专注于项目工作。

将每个轮次限制在特定的个人，有利于顺畅地与下一班次交接。在这种情况下，由两个人进行交接，而不需要大规模的运营团队会议。如果超过25%的团队必须专门从事单据任务和值班任务，在需要“救火”和缺乏自动化的情况下就会出现严重的问题。

团队经理应该成为运营轮次的一员。这种做法能够确保经理知道运营的负荷和发生的“救火”事件，还能确保不会因为疏忽而在运营组织中雇佣非技术背景的经理。

如果分类中的工作数量足够小，团队可以将值班和单据任务合并为一个工作岗位。有些团队可能需要在每个角色上分配多名人员。

项目工作最好在小规模团队中进行。个人项目可能使成员感觉孤立，或者使成员在没有建设性反馈的情况下工作，从而损坏团队。设计最好至少有一些同行评审。如果没有反馈，成员最终可能忙于他们觉得重要、实际上却没有多少好处的项目上。相反，大规模的团队往往因为缺乏共识而停滞不前。在这种情况下，专注于快速交付可以克服许多问题，使进度可见于项目成员、更广泛的团队和管理层，为项目提供帮助。经常收到反馈，就更容易修正航向。

8.6节中讨论的敏捷方法是组织项目工作的有效手段之一。

元工作

还有一些元工作：会议、状态报告、公司职能。这些工作通常会吞噬项目的时间，应该最大限度地减少。参见 Limoncelli 所著的《Time Management for System Administrators》(2005) 的第 11 章。

7.3.1 团队成员的工作日类型

现在，我们已经为团队的工作确立了组织原则，每个团队成员根据日期类型组织自己的工作：以项目为焦点的日子，值班日，单据任务日。

以项目为焦点的日子

运营人员的大部分工作日应该是项目日。确切地说，大部分时间应该花在开发自动化或者优化团队职责各方面的软件上。非软件项目包括指导新的启动工作，或者和利益相关方一起协商未来发行版本的需求。

通过单一的缺陷跟踪系统组织团队工作有个好处：减少花在检查不同系统状态上的时间。缺陷跟踪系统提供排定工作优先级和跟踪工作的简易手段。在典型的项目日中，团队成员首先检查缺陷跟踪系统，审核指派给他们的缺陷，或者审核团队成员可能需要采取措施的较高优先级未分配问题。

运营中的软件开发倾向于复制敏捷方法：不进行突然的大规模更改，而是随着时间推移用许多小项目演化系统。第 12 章更详细地讨论自动化和软件工程的主题。

不涉及软件开发的项目可能涉及技术工作。将服务移到新数据中心是高技术性的工作，因为不经常发生而无法被自动化。

运营人员倾向于不实际接触硬件，这不仅是因为虚拟机的大量使用，还因为即使位于数据中心的物理机器也距离很远。数据中心技术人员可以作为远程助手 (Remote Hand)，在必要时应用物理更改。

值班日

值班日花费在项目工作上，直到接收到警报，这种警报通常由短信息 (SMS)、文本消息或者传呼机发送。

一旦接收到警报，工作将转向该问题直到解决。一个问题往往有多种解决方案，通常包括快速而暂时性地修复问题，以及长期修复的其他方案。一般来说采用快速修复，因为将服务恢复到正常运营参数是重中之重。

解决警报之后，总是应该完成一些其他任务。应该以某种电子警报日志的形式分类和注解警报，以便发现趋势。如果采用快速修复，应该登记缺陷，请求长期修复。值班人员可以花一些时间更新这一警报的剧本条目，从而构建组织记忆。如果发生了用户可见的运行中断或者违反 SLA 的情况，应该撰写事后剖析报告，展开调查以确定问题根源。编写事后剖析报告、登记缺陷以及根源识别都是升级问题可见性、引起注意的手段。否则，我们

将持续地陷入临时的变通手段，情况永远不会变好。事后剖析报告（可能对技术性内容进行删节）可以与用户社区共享，建立对服务的信心。

在任何时候都为值班任务指派特定人员的好处是，团队的其余人员可以继续专注于项目工作。研究发现，软件开发人员生产率的关键是有较长时期不被打断。话虽如此，如果出现重大危机，值班人员将从项目中抽调人员协助。

如果班次轮换时间太长，值班人员将因为后续工作而超载。如果班次轮换太频繁，就没有时间完成后续工作。新项目和改善的许多好主意都是在应付警报的时候想出来的。在两个班次之间，应该有足够的时间从事这些项目。

第14章将更详细地讨论值班。

单据任务日

单据任务日花在来自客户的请求上。这里的客户指的是服务的内部客户，如使用服务API的其他服务团队，而不是来自外部用户的单据，那些项目应该由客户支持代表处理。

值班的预期响应非常快，而单据通常需要以天计算的响应时间。

典型的单据由关于服务的问题组成，可能导致对如何使用服务的相关咨询。单据也可能是激活某个服务的请求、问题报告或者用户遭遇的困难等。有时候，单据由自动化系统创建。例如，监控系统可能检测到某种不那么紧急、不需要立即响应的情况，可能会开立单据代替。

有些前一班次留下的长时单据可能需要跟进，往往采取一种策略，如果我们等待来自客户的响应，每3天会礼貌地“刺探”客户，确保问题没有被遗忘。如果客户等待我们的跟进，可能也有一种策略：紧急的单据有一个每天发布的状态更新，对于其他优先级的任务则采用更长的时间跨度。

如果单据到班次结束时没有完成，其状态应该包含在班次报告中，以便下一轮次的人员可以知道前一班次的结束位置。

指派某个人专门从事单据任务，这个人在响应单据时就会更加专注。所有单据可能经过筛选和优先级排定，有更多的时间分类单据，就更有可能发现趋势。通过批量处理类似的单据，可以实现更高的效率。更重要的是，专门处理单据的人应该有时间更深入地研究每个单据：更新文档和剧本、深入研究缺陷，而不是寻找肤浅的变通手段、修复复杂的不完整过程。单据任务不应该被视为琐事，而应该是减少团队所面临总体工作的策略的一部分。

每个运营团队都应该以消除开立单据的人作为目标，就像应该总是以自动化人工过程为目标一样。单据请求信息是文档应该改进的迹象。响应问题的最佳方式是在服务的FAQ或者其他用户文档中添加请求的信息，然后将用户引导到该文档。服务激活、分配或者配置更改的请求表明通过创建一个基于Web的门户或者API消除这类请求的机会。自动化系统创建的任何单据应该有对应的剧本条目，解释如何处理单据，包含指向缺陷ID的链接，请求改善自动化措施以消除开立这种单据的需求。

在值班和单据任务轮次结束时，当班人员常常用电子邮件向整个团队发送交接班报告。此报告应该指出任何注意到的趋势和传递给下一班次人员的任何建议或者状态信息。值班任务的交接报告还应该包含接收到哪些警报的日志和完成的响应措施。

在执行值班或者单据任务时，那就是你的主要项目。其他项目工作的完成是额外的收获。管理层不应该期望完成其他项目，也不应该因为属下专注于正常的事务而对其进行惩罚。当运营人员完成其值班或者单据任务轮次，不应该抱怨自己不能完成任何项目工作。可以说，他们的项目就是票据任务。

7.3.2 其他策略

组织团队工作还有许多其他手段。团队可以轮班，但是项目专注于特定的目标或者子系统，团队也可以减少“苦活”作为重点，或者专门留出特殊的时间用于减少技术负债。

焦点或主题

人们可以选择一类问题作为一两个月内的焦点，定期或者在当前主题完成时转换主题。例如，在某个主题开始时，可以选择一些安全相关问题，每个人都投入这一焦点任务，直到任务完成。一旦完成这些项目，就开始下一主题。常见的主题包括监控、特定的服务或子服务，或者特定任务的自动化。

如果团队的凝聚力较弱，这种做法可以帮助每个人感到似乎再次作为一个团队一起工作，也有助于改进生产率：如果每个人都已经熟悉了代码库的相同部分，就可以更好地相互帮助。

引入一个主题，还能提供某种动力。如果团队期待下一个主题（因为它更有趣、新颖或者令人愉快），他们就更有动力实现当前主题的目标，以便开始下一个主题。

减少苦活

苦活（Toil）指的是特别令人疲劳的手工作业。如果团队计算花在苦活上的时间，和常规项目工作做对比，这个比率应该尽可能低。管理层可以设置一个阈值，当比率超过 50%，团队应该暂停所有新功能的开发，致力于解决成为这么多苦活根源的大问题。

修复日

可以留出一天（或者连续几天）用于减少技术负债。技术负债（Technical Debt）是小规模未完成工作的累积。这些工作本身并不紧急，但是不断堆积使其开始成为一个问题。例如，在“文档修复日”中，每个人停下其他工作，专注于需要改进的文档相关缺陷。另外，可以宣布一个“修复周”，专注于将所有监控配置升级到特定标准。

团队往往将修复工作转变为游戏。例如，从一开始就发布任务（或者缺陷）清单。对于修复最多缺陷的人给予奖励，如果在整个公司范围内展开，参与或者完成最多任务的团队可以得到 T 恤。

7.4 虚拟办公室

许多运营团队在家中而不是在办公室工作。由于工作是虚拟化的，在必要时有远程助手操作硬件，我们可以在任何地方工作。因此，在任何场所工作是常见的现象，在必要时，团队在网络聊天室或者其他虚拟会议空间中会面，而不在实体会议室中开会。以这种方式工作时，团队的沟通必须有更明确的意图，因为大家并不是在办公室中“偶遇”。

任何不在办公室中工作的人有责任保持与团队的接触，是一种好的政策。他们应该定期清晰地通报其状态。反过来，整个团队有责任确保远程工作人员不觉得孤立。每个人应该知道其他团队成员正在从事的工作，并花时间与每个人讨论。有许多工具可以帮助实现这一目标。

7.4.1 沟通机制

网络聊天室常用于保持全天的联络。聊天室的记录应该保存并可访问，这样人们可以阅读可能错过的信息。有许多聊天室“机器人”（加入聊天室并提供服务的软件机器人）可以提供转录服务，将消息传递给离线成员，在交接班时发出通知，广播监控系统生成的警报。有些机器人提供娱乐服务：在 Google，有一个机器人记录谁接收的虚拟“击掌”最多。在 Stack Exchange，如果有人输入“不是我的错”，一个机器人就会注意到，从聊天室中随机选择一位用户，并通知他已经被随机地指定为替罪羊。

较高带宽的沟通系统包括语音和视频系统以及屏幕共享应用。带宽越高，就可以实现越高的沟通保真度。文本聊天不适合于表达感情，而语音和视频可以。在表达感情更重要时切换到更高保真度的沟通系统，尤其是密集、激烈的辩论开始时。

具有最高保真度的沟通媒介是现场会议。虚拟团队从定期的现场会议中得益甚多。每个人前往同一地方，参加为时数天的会议，集中讨论长期规划、团队建设和其他无法在网上解决的问题。

7.4.2 沟通策略

许多团队建立一个沟通协议，阐明在各种情况下使用的方法。例如，共同达成的协议是以聊天室作为主要沟通渠道，但是只用于临时性的讨论。如果在聊天室中做出一个决定或者需要公告，将通过电子邮件广播。电子邮件用于需要跨越值班轮次或者日期边界的信息。有持续效应的公告（如重大策略或者设计决策）必须在团队 Wiki 或者其他文档系统中记录（文档的创建必须通过电子邮件公告）。建立这种聊天 - 电子邮件 - 文档范型，最终能够减少沟通问题。

7.5 小结

运营不同于典型企业 IT，因为它的焦点是特定服务或者一组服务，对正常运行时间有

更高的要求。

运营团队对稳定性的渴望与开发人员对新代码投产的渴望之间存在紧张的关系。实现平衡的方法很多，大部分方法通过分担正常运行时间和新功能推出速度的职责，协调双方的目标。

分布式计算中的运营是大规模进行的，必须人工完成的过程无法实现伸缩性，不断地改进过程和自动化是必不可少的。

运营负责服务的生命期：启动、维护、升级和退役。维护任务包括紧急和非紧急响应。此外，相关的项目维护和演化服务也包括其中。

服务的启动和退役以及不经常执行的任务需要注意细节，这些细节最好通过使用检查列表确保。检查列表确保过去得到的教训发挥作用。

最能发挥运营人员时间效率的是将时间花费在过程的自动化和优化。这应该是他们的主要职责。此外，其他两种工作也需要注意。紧急任务需要快速响应。非紧急请求必须加以管理，以便排定优先级，及时解决。为了确保这些任务的完成，在任何时候运营团队中应该有一个人专注于紧急情况的响应，应该指派另一个人排定非紧急请求的优先级并加以解决。当团队成员轮流处理这些任务时，通过在整个团队中分担职责，得到了正确工作所需的专门资源，团队人员也可以避免过劳。

运营团队通常在远离运行服务的实际机器的地方工作。因为远程运营服务，他们可能任何有网络连接的地方工作。因此，团队往往在不同位置工作，在聊天室或者其他虚拟办公室中合作与交流。许多工具可以实现这种组织结构。在这样的环境中，根据所需沟通类型改变通信媒介很重要。聊天室对一般的沟通是足够的，但是语言和视频更适合于较为密集的讨论。电子邮件在需要记录沟通，或者联系目前不在线的人时较为合适。

练习

1. 什么是运营？它的主要职责范围是什么？
2. 分布式计算中的运营与传统桌面支持或者其他客户 - 服务器支持有何不同？
3. 描述和你经历的服务关联的服务生命期。
4. 7.1.2 节讨论变化 - 不稳定循环。画一个系列图，其中 x 轴是时间，y 轴是稳定性指标。每个图表应该表示两个月的项目时间。
每周一试运行引入不稳定性（9 个缺陷）的重大软件发行版本。在周二到周五，团队有机会试运行“缺陷修复”发行版本，每个版本修复 3 个缺陷。画出如下情况的图表：
 - a) 没有缺陷修复版本。
 - b) 每个重大发行版本之后有两个缺陷修复版本。
 - c) 每个重大发行版本之后有三个缺陷修复版本。
 - d) 每个重大发行版本之后有四个缺陷修复版本。

DevOps 文化

DevOps 就是绝望的对立面。

——Gene Kim

本章介绍的是被称作“DevOps”的文化和一组实践。在 DevOps 组织中，软件开发人员和运营工程师作为一个团队协同工作，分担网站或者服务的职责。这和其他组织形成对比，在那些组织中，开发人员和运营人员各自为战，目标常常发生冲突。DevOps 是运营 Web 服务的现代化方法。

DevOps 将某些文化和态度转变与一些常识性过程结合起来，最初的基础是在运营中使用敏捷方法，结果形成一组合理化的原则和过程，能够创建可靠的服务。

附录 B 证明云或者分布式计算（Distributed Computing）是硬件经济学的必然结果。DevOps 是这种环境中高效运营需求的必然结果。

如果硬件和软件有足够的容错性，剩下的问题就是人。Oppenheimer 等人撰写的论文《Why Do Internet Services Fail, and What Can Be Done about It?》（2003）提出了这样一种认识：如果 Web 服务要在未来取得成功，必须改善运营方面：

我们发现：（1）在 2/3 的服务中，运营人员的错误是最大的故障起因；（2）运营人员的错误往往需要很长时间才能修复；（3）配置错误是最大的运营人员错误类别；（4）定制前端软件中的故障很严重；（5）更大规模的在线测试和组件故障的更全面揭示与检测，能够降低至少一个服务中的故障率。

换言之，技术已经变得很可靠，剩下的问题在于管理技术的过程中，我们需要更好的实践方法。

改善大规模运营的权威方法是通过 W. Edwards Deming 的“Shewhart 循环”（Deming，

2000) 或者“精益制造”(Spear & Bowen, 1999) 等“质量管理”技术。DevOps 的关键原则是这些原则在 Web 系统管理上的应用。《The Phoenix Project》(Kim, Behr & Spafford, 2013) 一书以虚构故事的形式解释了这些原则, 讲述了一个团队学习这些原则, 改造一个失败 IT 组织的故事。

8.1 什么是 DevOps

DevOps 是文化和实践的结合——系统管理员、软件开发人员和 Web 运营人员都为 DevOps 环境做出贡献。利用 DevOps, 系统管理员和开发人员分担服务及其可用性的职责。DevOps 使开发人员 (dev) 和系统管理员或运营人员 (ops) 都对正常运行时间负责任, 协调两者的任务优先级。DevOps 还将开发、测试和生产等各种环境置于软件版本管理和控制之下。

在最基础的层面上, DevOps 旨在打破竖井, 消除影响组织开发-运营交付生命期的瓶颈和风险, 目标是使规格中的变化快速可靠地转化为面向客户环境中的运行特征 (Edwards, 2012)。

DevOps 是运营中的新兴领域。DevOps 的实践通常发生在 Web 应用和云环境中, 但是它的影响遍及所有行业的各个角落。

DevOps 是关于运营改善的学科。Theo Schlossnagle (2011) 说 DevOps 是“世界的运营主义”。越来越多公司对业务的运营方面给予更大的重视, 这是因为关心项目的总拥有成本 (TCO) 而非初始购买成本已经成为趋势, 而且实现更高可靠性和快速变化的压力与日俱增。变化的能力是增进效率和引入新功能及创新所必需的。传统上将变化视为潜在的不稳定因素, 而 DevOps 说明, 基础设施的变化可以快速而频繁地进行, 同时改善总体的稳定性。

DevOps 不是一个职衔, 你不能雇佣一位“DevOp”。它也不是一个产品, 你无法订购“DevOps 软件”。有些团队和组织展现了 DevOps 文化和实践。许多实践得到了某个软件包的帮助, 但是无法购买一个盒子, 按下 DevOps 按钮, 像变魔术一样拥有 DevOps。Adam Jacob 在 Velocity 2010 (Jacob, 2010) 上题为“Choose Your Own Adventure”的开创性演讲说明了 DevOps 不是一个职位, 而是形成某种文化的所有活动。在这种文化中, 每个涉身其中的人都知道整个系统的工作原理, 每个人都清晰地知道自己呈现的商业价值。因此, 可用性成为了整个组织的问题, 而不仅仅是系统管理员的事。

DevOps 不仅仅与开发人员和系统管理员相关, 在题为“DevOps is not a technology problem. DevOps is a business problem”的博客帖子中, Damon Edwards (2010) 强调, DevOps 与整个组织的协作和优化有关。DevOps 可以扩展, 协助从思路到客户的整个过程。它不仅仅是利用巧妙的新工具, 实际上, 也不仅仅和软件有关。

创建 DevOps 环境所涉及的组织更改最好通过和传统软件开发方法的比较来理解。DevOps 方法的发展是因为传统方法在开发定制 Web 应用或者云服务产品时的缺点, 以及满足这些环境中更高可靠性要求的需要。

8.1.1 传统方法

对于计算机商店中放在封套中销售或者从互联网下载的软件，开发者的工作在软件完成并交运时就已结束。运营问题只直接影响客户，开发人员游离于运营之外。运营问题至多以缺陷报告或者改进申请的形式反馈给开发人员，但是开发人员并不直接受到自己编写的代码引发的运营问题的影响。

传统软件开发使用瀑布方法（Waterfall Methodology），每一步——收集需求、设计、实现、测试、验证和部署——由不同团队完成，每个步骤和其他步骤隔离。每个步骤（团队）产生某种可交付物，移交给下一步（团队）。

这种方法称作“瀑布开发”是因为步骤看起来像是多级的瀑布（参见图 8.1）一样。信息像水一样向下流动。

至少在设计完成之前，了解系统的运营需求是不可能的，在许多情况下甚至还要推迟到系统部署并广泛使用之后。因此，运营需求在需求收集阶段并没有得到考虑，在这个阶段之后功能已经“锁定”。于是，这种方法的结果是，考虑运营需求时已经太晚，无法采取任何补救步骤。

在瀑布方法中，系统管理员仅涉及软件部署工作，因此只负责运营和满足正常运行时间的要求。系统管理员没有机会影响软件的开发，以更好地满足自身需求。在许多情况下，他们不直接和软件开发人员接触。

采用 Web 业务模式或者 Web 形象至关重要的公司需要开发自己的定制软件。传统上，软件开发人员和系统管理员即使在同一家公司工作，互动也很少。他们在“竖井”中工作，每个小组不知道其他小组的关注点，双方也都看不到“全局”。在组织结构图中，他们的层次结构可能只有到了 CEO 级别才有交点。软件开发人员继续独立开发软件，对未来的使用毫无感觉，系统管理员则继续苦苦挣扎，用充满缺陷的软件去满足高可用性需求。

在这种情况下，运营通常通过临时解决方案改进，采用权宜之计并在有限的范围内优化。这种方法妨碍了运营效率、性能和正常运行时间的改善。唉，瀑布方法毕竟是来自于我们儿时的方法。^①

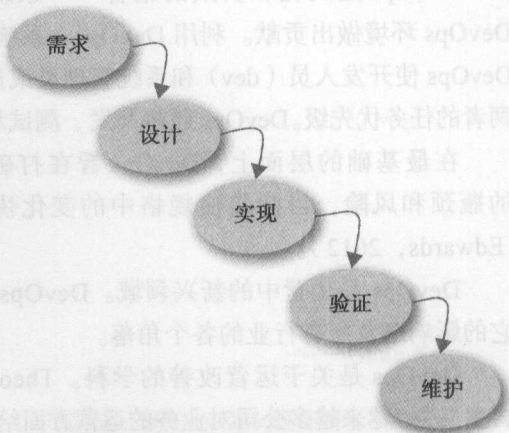


图 8.1 瀑布方法：信息向下流动，单向信息流是 DevOps 的对立面

① 真的是这样吗？Royce 在 1970 年的论文中“发明”了瀑布模型，文中他已经发现了这些问题并提出批评和改进建议。他写道，因为“设计迭代从未受到后续步骤的限制”，所以“充满风险并且容易招致失败”。Royce 提出的替代方法类似于现在的敏捷方法。遗憾的是，好几代软件开发人员在瀑布项目中遭受苦难，我们只能将此归咎于人们没有完整地读过篇论文（Pfeiffer 2012）。

8.1.2 DevOps 方法

采用基于 Web 的业务模型的公司发现，传统软件方法无法满足很高的可用性要求。对于高可用性来说，运营问题是关键，所以这些公司需要新的方法，将开发和运营紧密结合起来。结果是发展出了一组新的方法，并发明了“DevOps”这一术语来描述它们（DevOps 的历史沿革参见附录 B）。

基于 Web 的公司推出新功能通常比销售套装软件的公司更频繁。例如，由于最终用户很容易从一个 Web 搜索引擎切换到另一个，这些公司必须不断改善其产品，以维持客户基础。而且，Web 公司发行新版本更频繁还因为它们能够做到——它们没有必要制造物理媒体并将其分发给客户。采用传统套装软件的方法，每个新发行版本都被视为对稳定性的不良影响——是新的未知缺陷的来源。

在 DevOps 环境中，开发人员和系统管理员分担满足正常运行时间要求的责任，也分担值班任务。开发人员的既得利益在于，可以确认他们的软件满足网站的高可用性要求。开发人员协助创建运营策略，而运营人员与开发人员紧密合作，提供实现和开发意见。开发和运营都由一个团队处理，开发人员和系统管理员参与所有阶段，共同对最终成果负责。开发周期应该是一组产生最终产品（服务）的无缝过程。没有了“他们”的概念，比如“移交给他们”。只有“我们”——致力于产品的团队。团队成员大都是具有深厚专业知识的通才。

大部分 DevOps 专注于清晰的业务目标，如伸缩性、效率和高正常运行时间。DevOps 强调任何人和过程胜过特殊工具的使用，紧密协调运营和业务需求，从而达到和客户需求的一致。

通过要求运营、开发和业务部门协同工作，DevOps 中的运营过程变成共同的职责，可以更快、更有效地响应所运营服务的需求。DevOps 方法的结果是更长的正常运行时间和更低的运营成本。

8.2 DevOps 的 3 条道路

“DevOps 的 3 条道路”是改进运营的一种策略。它描述了组成 DevOps 过程、规程和实践框架的价值观及哲学。“3 条道路”策略因为 Kim 等人（2013）的《The Phoenix Project》一书而流行，借鉴了“精益制造”（Spear & Bowen, 1999）和丰田制造体系的持续改善模型。

8.2.1 第一条道路： workflow

工作流着眼于过程从头到尾的正确性，以及过程完成速度的提高。过程是一个价值流（Value Stream）——为企业提供价值。速度被称作流速（Flow Rate）或者简称流（Flow）。

如果过程中的步骤列在时间轴上，可以看成从左到右改进过程。左侧是业务（开发），

右侧是客户（运营）。

例如，软件发行过程有多个阶段：代码提交到代码存储库，单元测试，打包，集成测试，部署到生产环境。为了强调过程自始至终保持正确，采取如下措施：

- ❑ **确保每个步骤以可重复的方式进行。**用可重复过程代替偶然性和临时性的步骤。
- ❑ **绝不将不合格品传递给下一步。**测试尽早进行，而不仅在最终产品上。每一步都有验证或者质量保证检查。
- ❑ **确保局部优化不会导致整体性能降级。**例如，不打包软件而在每一步中从源存储库提取软件可能更快。这节约了开发人员的时间，因为消除了一个步骤。与此同时，其余步骤是否都使用完全相同的代码这一点上产生了不确定性，从而导致混乱、增加错误。因此，这会导致总体退化，我们不应该这样做。
- ❑ **改进工作流程。**既然各个步骤以可重复的方式进行，过程可以分析和改进。例如，可以自动化步骤以提高速度。另外，有些步骤中工作多次重新进行，这些重复的工作可以消除。

8.2.2 第二条道路：改进反馈

信息（投诉或者申请）向上游或下游传达，会确立一个反馈循环。扩大反馈循环意味着确保从左（开发）到右（运营）学习到的经验教训会被传递回左侧，并再次通过该系统。反馈（关于问题、关注点或者潜在改进的信息）是可见的，而不会被隐藏。当我们从左向右推进工作时，学习到一些经验教训。如果最后这些教训被抛弃，就失去了改进系统的机会。相反，如果这些经验教训得到放大并且可见，就可以用于改进系统。

继续我们的软件发行示例，强调反馈循环的放大：

- ❑ **理解和响应所有内外部客户。**除了过程结束时显而易见的“客户”之外，每个步骤都是前一步骤的客户。理解客户意味着理解后续步骤的需求。响应意味着客户有沟通的途径，以及确保请求得到响应的手段。
- ❑ **缩短反馈循环。**缩短反馈循环意味着使沟通尽可能直接。信息沟通所经过的阶段越多，效率就越低。如果反馈信息被交给一位经理，由其打印之后交给一位副总裁，副总裁再传达给一位经理，经理告诉工程师，那么步骤就太多了。如果遇到问题的能够直接和解决问题的人沟通，循环就足够简洁了。
- ❑ **放大所有反馈。**这一点的对立面是有人注意到一个问题，用自己的变通手段加以处理。这个人可能认为自己是解决问题的英雄，事实却是，这个人隐藏了问题，阻碍了问题的解决。放大反馈使问题的能见度更高，这种做法可能很简单——只需要填写一份缺陷报告，也可能很引人注目——停止过程，直到管理层做出如何继续的决策。当所有反馈都和盘托出，我们就有了改进过程所需的最多信息。
- ❑ **将知识嵌入所需的地方。**配置信息或者业务需求等特殊知识通过使用合适的文档“嵌入”过程中，并通过源代码控制系统管理。当你在过程中从左向右移动时，所

需细节在每个阶段都可以找到，不需要跳出循环获取。

8.2.3 第三条道路：持续试验和学习

第3条道路包括创造一种文化，鼓励每个人尝试新事物。这是创新的需要。在第3条道路中，每个人都理解两件事：（1）我们从试验和冒险发生的失败中学习；（2）掌握一门技能需要反复实践。

在我们的软件发行示例中这意味着：

- **创造奖励冒险的仪式。**尝试新事物，即使失败，如果得到了宝贵的经验教训，在审核时也应该奖励。
- **管理层为项目分配改进系统的时间。**“技术负债”的积压是个重要问题。应该分配资源，修复放大反馈时提交的缺陷，错误不能重复出现。
- **在系统中引入故障以增强弹性。**在应急演练（第15章中讨论）有意地造成机器停机或者网络中断，确保冗余系统起作用。
- **尝试“疯狂”或者鲁莽的事。**例如，可以尝试将流程时间从一周缩短为一天。

当团队能够识别其“价值流”（企业所依赖的过程）并应用 DevOps 的3条道路时，不仅能使过程变得更好——公司也会更加珍视 IT 团队。

8.2.4 小批次更好

DevOps 的另一条原则是小批次更好。

小批次意味着发行许多个功能较少的小发行版本，而不是发行少量具有大量功能的“大”发行版本。大规模、低频率的发行风险很大。大量新功能使代码中缺陷的跟踪变得很困难。功能可能相互干扰，造成新的缺陷。为了降低总体风险，发行许多小的版本，每个版本仅包含少数功能，这样的做法更好。

这种模式的第一个好处是每个新发行版本较小，更容易隔离缺陷。第二个好处是减小代码延迟。代码延迟是从第一次“签入”（check-in）到投产（可以为你赚钱）的速度。从财务的角度，代码越快投产，意味着越快产生投资回报（ROI）。最后，小的批次意味着过程在许多次迭代上完成，也就是说进行了更多实践，这样就有更多的机会改善这一过程，降低了风险。

周转率（Velocity）是每个月中交运产品的次数，高周转率和低延迟通过发行小的批次实现。

小批次原则是违反直觉的，因为人们倾向于避免高风险行为。在生产环境中部署软件涉及风险，因此企业传统上最大限度地降低部署频率。虽然这让他们感觉更好，但是实际上是搬起石头砸自己的脚，因为他们所做的部署规模更大、风险也更大，实施部署的团队在下一次部署时已经久疏战阵。

这条原则适用于部署和其他任何涉及频繁更改的过程。

8.2.5 策略的采用

采用上述 3 条道路的第一步是识别团队的价值流——为企业服务的过程，或者企业请求的过程。

多次经历每个过程，直到可以自始至终无失败地完成。过程不一定是最优的，但是每个步骤必须清晰定义，能以可重复的方式进行。也就是说，经过合格培训的人应该可以完成这些步骤，结果应该和另一个受过合格培训的人一样。现在定义过程。

定义过程之后，扩充反馈循环。也就是说，确保每个步骤都有升级问题能见度的手段，使这些问题可以得到解决，不会被忽视。收集步骤长度、频率、失败率的指标，使这些数据可用于涉及的所有人。

这些反馈用于优化过程，找出最容易出错、不可靠或者缓慢的步骤。替代、改善或者消除这些步骤。效率最低的两个环节是返工（修复错误）和多余的工作（可以合并的重复劳动）。

每个过程都有瓶颈——等待其他依赖因素而使工作延迟的地方。在瓶颈处投入精力改善是最有好处的。实际上，在任何其他地方优化都是浪费精力。在瓶颈之上，没有完成的工作会堆积起来。在瓶颈之下，工作者却无事可做。优化瓶颈之上的步骤只会使堆积的工作变得更多，在瓶颈之下优化，改进的则是利用率很低的步骤。因此，修复瓶颈是唯一合乎逻辑的做法。

要实现这一切，需要创新的文化和冒险精神。必须鼓励冒险，并接受失败。实际上，一旦采用“DevOps 的 3 条道路”实现顺畅的最优化过程，就应该在系统中引入缺陷，验证它们是否能够被检测和处理。通过这样接受失败，我们就能从优化迈向弹性。

8.3 DevOps 的历史

“DevOps”一词是 Patrick Debois 于 2008 年提出的。Debois 注意到，有些网站已经将系统管理发展为某种完全不同的实践。也就是说，它们已经独立地得出结论：在开发和运营相互协作时，网站可以运行得更好。Debois 认为，将这些人聚集到一起分享各自的经验是很有价值的。他于 2009 年从比利时开始发起一系列小型会议，称作“DevOps Days”。这个名称来自将开发人员（Dev）和运营人员（Ops）聚集在一起的概念。

DevOps Days 取得了很大的成功，促进了“DevOps”一词的流行。这些交流在邮件列表和博客上持续了下去。2010 年 5 月，John Willis 和 Damon Edwards 启动了 DevOps Cafe 播客，这个播客很快成为了交换和讨论 DevOps 思路的场所。主题标签“#devops”成为 DevOps 追随者们在 Twitter 上的标志，在当时 Twitter 还是相对新颖的服务。2011 年的 USENIX LISA（大安装系统管理）会议选择 DevOps 作为主题，此后该会议将 DevOps 作为一个焦点。

8.3.1 演变

有些从业者认为, DevOps 是让系统管理员与参与敏捷开发周期的开发人员一起工作, 对系统工作采用敏捷技术的一种合乎逻辑的发展。虽然, DevOps 中常常使用敏捷工具, 但是敏捷仅仅是应用 DevOps 原则的许多途径中的一种。结对编程和 Scrum 团队等技术对于创建 DevOps 环境并不是必要的, 但是坚持敏捷的某些基本原则是绝对必需的。

另一些从业人员则认为, DevOps 是由于 Amazon AWS 和 (后起的) 类似服务流行, 开发人员自己从事系统管理的合乎逻辑的演变。换言之, 开发人员再造了系统管理。过去, 安装一台新机器需要系统管理的技能和培训。现在, 开发人员用 API 调用分配虚拟机。不需要全职的系统管理员, 开发人员越来越多地学习系统技能, 并引入了他们对自动化任务的嗜好。通过使部署和测试成为开发周期中的编码部分, 他们重新强调了可重复性, 造就了本章讨论的许多种技术。

还有一些从业人员反对上述看法, 认为有些系统管理员一直强调自动化, 但是 Web 环境之外的管理层并不珍视这些技能。从他们的角度看, DevOps 是由专注编码技能、开始和开发人员在部署及代码测试上展开合作的系统管理员推动的。将这些步骤纳入开发周期, 形成了和开发人员更紧密的联系, 他们像亲密无间的小组一样协同工作, 实现增加正常运行时间和无缺陷部署的共同目标。更有讽刺意义的是, DevOps 可以视为由系统管理员重塑的系统管理方法, 最终只有在管理层的支持下才能正确进行。

8.3.2 网站可靠性工程

差不多在同一时期, Google 等公司开始对其互联网系统管理实践采取更为开放的态度。Google 承认系统管理的所有职能 (从容量到安全性) 对于网站的可靠性至关重要, 将系统管理演化为网站可靠性工程 (SRE) 的概念。从 2005 年起, Google 的 SRE 模式将该公司的开发人员和运营工程师组织起来, 分担可靠性和性能的责任。SRE 模式可以看作大规模的 DevOps: 如何让 10 000 名开发人员和 1000 名 SRE 协同工作? 首先, 每个产品或者基础设施组件都由一个小型团队负责。对于关键性和高可见性的系统, 开发人员和 SRE 按照 DevOps 模式的安排一起工作。遗憾的是, 没有足够的 SRE。因此, 大部分此类团队由使用 SRE 所开发工具的开发人员组成。为此, 设计了一些工具, 使运营成为开发人员的自助式服务, 准许开发人员进行自己的运营工作。SRE 构建专门设计的工具, 便于在没有高级知识的情况下实现高质量的结果。

DevOps 快速地从一个小众技术扩展为可以应用于企业和行业系统管理的技術。关于 DevOps, 没有任何以 Web 为中心的独特内容。Marc Andreessen 曾经说过一句著名的话, “软件吞噬世界” (Anderson 2012), 由于这种情况的发生, 社会的各个层面都需要很好的运营。因此, DevOps 将适用于计算的所有方面。

8.4 DevOps 价值观和原则

DevOps 可以大致分为 4 个主要的实践领域 (Kartar 2010):

- 关系
- 整合
- 自动化
- 持续改进

8.4.1 关系

在传统环境中, 工具和脚本被视为运营维护的主要焦点。DevOps 更重视团队之间的关系以及组织中的各种角色。开发人员、发行管理人员、系统管理员和经理们——都需要紧密协调, 以实现共同的目标——高可靠性和持续改进的服务。

DevOps 环境中的关系如此重要, 以至于有一句常见的格言: “人胜过过程, 过程胜过工具”。只有一致地让合适的人执行合适的任务, 才能够创建工具自动化运营职能。DevOps 的关键定义原则之一是, 将焦点放在人和过程上, 而不是脚本的编写上, 然后确定谁、什么时候应该去运行脚本。

8.4.2 整合

确保过程跨越团队实现整合, 是打破竖井工作的一部分。在 DevOps 环境中不将运营任务仅仅看作遵循脚本的工作, 而将它们视为将工具、数据与人为过程 (如同行审核或者协调会议) 相结合的端到端过程。过程必须跨越职责领域相互联系, 以交付端到端的功能性。

负责服务运营不同部分工作的社区之间的整合也是 DevOps 的一个关注点。评估环境中 DevOps 文化的快速途径之一是询问系统管理员, 他们经常和谁共进午餐。如果答案是“大部分是系统管理员”, 而很少和来自软件开发人员、Web 运营、网络或者安全性团队的人一起, 这就是团队整合尚未实现的一个信号。

8.4.3 自动化

在自动化的支持下, DevOps 坚持简洁性和可重复性。配置和脚本被作为源代码处理, 并置于版本控制之下。在理解整个过程之后, 源代码的构建和管理也最大限度地脚本化。

简洁性增进沟通的效率和速度, 避免混乱, 还能够节约培训、文档和支持的时间, 目标是设计简单、可重复、可重用的解决方案。

8.4.4 持续改进

每次执行某个过程, 目标都是使其能够更可靠地重复、更实用。例如, 每次出现故障时, 都在发行过程中增加测试, 检测这种故障模式, 避免另一个具有同样缺陷的发行版本

进入下一步。另一个例子是，通过在工具中处理更多的边缘情况，改进需要偶尔进行人工干预的过程，直到最终不再需要人工干预。

采用端到端的视角，我们往往会发现消除过程和工具的机会从而简化系统。通过查找根源修复问题，而不是进行降低全局性能的局部优化。

Kartar (2010) 生动地总结了所需要的思维体系：

将你的过程当成应用程序对待，在其中植入错误处理。你不能预测每一个…陷阱…但是可以确保在遇到陷阱时过程不会偏离方向。

8.4.5 常见的非技术性 DevOps 实践

DevOps 这一术语包含了许多非技术性实践。并不是所有 DevOps 组织都使用全部此类方法。实际上，重要的是从中选择，使用必需的方法，而不是为了全面性而盲目地遵循所有实践。这些实践都是“人为过程”，更有技术性的实践将在下一节介绍。

- ❑ **及早协作和参与：**运营人员加入开发规划会议，开发人员能够全面接触运营监控。伸缩性架构等关键问题在规划阶段联合开发（第 5 章）。
- ❑ **新功能审核：**运营人员参与设计阶段，并就可运营性的最佳实践指导开发人员，而不是在事后介入。服务的关键监控指标通过协作定义。草拟部署细节，使部署代码和测试的开发成为主要开发工作的一部分（第 2 章）。
- ❑ **分担值班任务：**这些任务不仅包括开发人员和运营人员分担“传呼机”任务，还要分担值班任务中的趋势研究——例如，每周召开会议审核 SLA 依从性和任何运行中断。开发人员可以全面访问所有监控输出，运营人员可以全面访问所有构建 / 部署输出。这样，每个人都得到完整的授权，可以研究值班或者故障分析中发现的任何问题。
- ❑ **事后剖析过程：**除了定期的运行中断及趋势研究例会之外，还应该对每次运行中断进行全面的事后剖析或者故障分析。重复出现的小故障模式可能指向过程中较大的缺陷。事后剖析的发现——特别是，改正问题所需的任务——应该加入当前开发的积压工作，并相应地排定优先级。
- ❑ **“游戏日”演练：**有时候称作“应急演练”，这是为了测试故障切换和冗余性的有意之举，通过按照规划触发服务中断进行。团队随时待命，确保“正确的事情”发生，并人工修复没有解决的问题。只有引发故障，才能真正测试服务组件失效时发生的情况。游戏日演练的一个简单例子是定期随机重启选择的机器，确保所有故障切换系统正常运作。
- ❑ **错误预算：**追求完美会阻碍创新，但是过多的创新意味着冒太大风险。Google 的错误预算等系统可以达到两者的平衡。每月允许一定的停机时间（预算），在预算耗尽之前，开发人员可以根据自己的希望发行尽可能多的版本。一旦预算耗尽，他们在这个月的余下时间里只能进行紧急的缺陷修复。为了保持错误预算，他们可以投

入更多时间进行测试，构建确保成功发行的框架。这协调了运营和开发人员的优先级，帮助他们更好地协作。完整的描述参见 19.4 节。

8.4.6 常见的技术性 DevOps 实践

DevOps 从根本上说是一个结构化、有组织的范型。但是，为了实现 DevOps 的目标，已经采用或者开发了一系列技术实践。同样，并不是每个 DevOps 组织都采用这些做法。这些实践是工具箱中的工具，应该选择最适合你的情况的那些工具。

- ❑ **相同的开发和运营工具链：**开发和运营最好尽可能使用相同的工具，使用相同的“语言”。简单的做法是为开发和运营 / 部署问题使用相同的缺陷跟踪系统。另一个例子是使用统一的源代码管理系统，不仅存储产品的源代码，还存储运营工具和系统配置的源代码。
- ❑ **一致的软件开发生命期（SDLC）：**将应用程序本身和部署 / 运营代码一起置于相同的 SDLC，是保持两者同步的关键。“将皮球踢给部署人员”是 DevOps 所摒弃的做法，DevOps 中开发和运营是紧密结合的。部署工具和应用程序本身共同开发和测试，遵循共同的发行周期。
- ❑ **托管配置和自动化：**服务所需的所有应用程序的配置文件都置于源代码控制之下，和其他代码实施相同的更改管理。所有自动化脚本也是如此。
- ❑ **基础设施即代码：**有了软件定义的数据中心（即虚拟机），可以将整个基础设施的描述作为代码，在版本控制下维护。基础设施即代码在 10.6 节中进一步说明。
- ❑ **自动化配给和部署：**部署过程的每一步都自动化或者脚本化，可以触发从自测试直到部署的构建，或者通过单独的构建命令触发部署。
- ❑ **脚本化的数据库更改：**数据库更改也可以作为代码处理，而不是人工操纵数据库模式。这些更改进行脚本化、测试、版本控制，并发布到预演环境中。
- ❑ **自动化构建与发行：**构建循环的输出是一组有效的应用程序和部署对象，可以部署到预演环境中。构建有生成文件或者其他配置文件，将构建视为一系列相互依赖和需要履行的协议，可以通过登录或者特定的命令触发。手工的构建装配阶段不利于运营的可重复性和简易性。
- ❑ **发行载体包装：**如前所述，构建循环创建应用程序包以方便部署。构建的最终产品不需要为了准备部署而手工打包，也不需要在使用前通过访问存储库软件部署到实际系统或者在每台主机上编译。
- ❑ **抽象化管理：**抽象化管理在高层次上描述系统管理，由自动化系统决定给定操作系统所需执行的合适步骤。因此，我们用于配给新用户的配置文件中指令可能是“创建用户”，而不是 Linux 所需步骤（“将这行附加到 /etc/passwd，这行附加到 /etc/shadow，这行附加到 /etc/group”）或者 Windows 所需步骤（“在活动目录中创建用户”）。通过使用工具完成初始设置，我们简化了目标和配置之间的接口。这一领域

的常用工具包括 CFEngine、Puppet 和 Chef。

8.4.7 DevOps 发行工程实践

某些发行工程实践已经和 DevOps 实践紧密相关。发行工程是获得源代码形式的软件、构建、打包、测试、现场部署的过程。

虽然本身不是 DevOps 实践，这些开发实践对实现高效运营有很大帮助（DevOps-Toolchain 2010）。第 9、10、11 章将更详细地讨论这些实践：

- ❑ **持续构建：**对于每次更改，都尝试编译代码库、生成打包软件。这样做可以尽快发现与构建相关的问题。
- ❑ **持续测试：**软件在每次代码库更改时以自动化方式测试，这样可避免问题嵌入系统。
- ❑ **自动化部署：**对在测试和实际环境中使用的软件部署过程进行自动化。
- ❑ **持续部署：**随着构建、测试和部署的全面自动化，是否部署特定发行版本的决策也实现了自动化。每天在现场部署多个发行版本。
- ❑ **自动化配给：**CPU、存储、内存和带宽等附加资源根据预测性的模型分配。当系统检测到需要更多资源，就分配这些资源供系统使用。

8.5 向 DevOps 转化

实施上述建议之前，开发人员和运营人员必须开启对话。构建小组之间的桥梁必须从形成学术联系开始。最好的方式往往是远离办公室，来一杯啤酒或者其他饮料。此时运营和开发人员真正开始相互交流、分享看法、找到采用 DevOps 方法的共同点。在 DevOps Cafe 播客上的一次采访中，Jesse Robbins 指出，在薯条和饮料上花费 50 美元，可能是一些公司有史以来最佳的投资（Willis & Edwards 2011）。

在传统的非 DevOps 组织中采用 DevOps 原则时，重要的是慢慢地开始。首先采用少数新方法，在这些方法得到团队的支持之后再增加更多的方法。这种转化包括 3 个基本阶段。

首先，向整个项目团队公开运营反馈。这可以采用多种形式，最常见的是组织向开发人员开放监控，要求他们共同进行问题或者故障根源分析，识别重复出现的问题并合作提出解决方案。

其次，开始在运营中嵌入产品知识。邀请开发人员参加关于部署和维护的关键运营会议，并设定下班后联系开发人员的问题升级路径。

最后，在所有项目阶段中输入运营知识。这包括让运营人员参与每日或者每周状态审核会议，参与产品积压工作优先级排定，成为规划会议的全面合作伙伴。

8.5.1 准备开始

为了开启 DevOps 关系，必须首先离开电脑，开始面对面的讨论。从哪里开始？首先是

你所支持产品的开发人员、产品经理和产品团队的其他成员。

最好选择容易接近的人，也就是和你已经有某种亲密关系的人。你可以安排会议，或者简单地某个下午出去喝杯咖啡。

最初的谈话应该和可以解决的共同问题相关。作为谈话的一部分，解释紧密协作可能带来的改进，如改进发行效率，或者运营部门更好地响应开发人员的需求。

在讨论可以解决的问题时，你会发现联合 DevOps 项目的一个好的起点。选择对开发团队有明显好处的问题，而不是主要以运营为焦点的问题。最好从互惠的角度描述该项目。

作为项目协作的一部分，要形成和开发及产品团队召开定期会议的习惯。参加规划会议，并邀请他们参加你的规划会议。

成功完成启动项目时，找出另一个可以协作的项目。好的候选项目是同时影响开发和运营的过程，例如试运行或者工具链建设。来自开发、运营和产品团队的多个利益相关方的参与，是建立关系的好手段。

同样，计量 DevOps 成功的简单手段之一是提出“你和谁共进午餐？”的问题。如果你经常和来自开发、网络、发行或者类似小组的人员共进午餐，就很有可能正在进行 DevOps 工作。

8.5.2 企业层面的 DevOps

切换到 DevOps 环境的下一阶段是让管理层支持 DevOps 思想。真正的 DevOps 环境往往涉及组织结构图的更改，打破开发和运营之间的隔阂。组织结构需要培育开发和运营之间的紧密关系。理想情况下，将开发和运营置于同一管理团队、相同的副总裁或者某些类似领导人之下。还要尝试让各个小组在相同地点办公，以便相互靠近，或者至少处于同一建筑物或者时区。这类改变通常会展现出全新的效能水平。获得支持相当困难，管理层需要用业务术语解释 DevOps 价值观。

DevOps 并不止于开发人员和运营人员之间的协作，对整个组织链条都有作用。最近，本书的作者之一目睹了一个内部项目，开发人员、运营人员、产品管理人员和法律部门在该项目中一起工作，创建处理一组严格约束的解决方案。来自法律部门的人员对于集中于问题的协作和投入程度感到惊讶。该项目避免了购买昂贵的、需要配置和外部管理人员的第三方系统。

DevOps：不仅用于 Web

Gruver、Young 和 Fulghum (2012) 所著的《A Practical Approach to Large-Scale Agile Development: How HP Transformed HP LaserJet FutureSmart Firmware》描述了 DevOps 在 HP LaserJet 软件上的应用。结果是，开发人员进行人工测试上花费的时间更少，这样就使他们有更多的时间开发新功能。他们的成功故事在 DevOps Cafe 播客的第 33 集中总结 (Willis、Edwards & Humble 2012)。

8.6 敏捷和持续交付

DevOps 是被称作“敏捷”的软件开发方法和称作“持续交付”的实践的自然发展成果。虽然本书不是关于这些主题的，简单地考察敏捷和持续交付有助于说明许多 DevOps 的起源。涉及的原则可以直接转换为 DevOps 实践，作为 DevOps 思想体系的强大基础。

8.6.1 什么是敏捷

敏捷是一组软件开发原则，源于异军突起的各种代表性非传统软件方法（如极限编程、Scrum 和语义编程等）。它们自称为“敏捷联盟”并创作了很有影响的“敏捷宣言”（Beck 等人，2001）。

敏捷宣言

- 个人和互动优于过程和工具
- 可工作的代码优于全面的文档
- 客户协作优于合同谈判
- 对更改的响应优于遵守计划

在《敏捷宣言》的最后，作者们补充道，“虽然右侧的项目有其价值，我们更重视左侧的项目。”敏捷方法强调业务目标和开发团队之间的直接联系。开发人员紧密地与产品负责人协作，构建符合特定业务目标的软件。开发的瀑布方法被位于结对编程（两位开发人员协作编码）或者 Scrum（整个团队投入为期 1~4 周的“冲刺”，致力于排定优先级的积压功能）回避。称为“用户故事”的简单陈述提供了软件开发的需求——例如，“作为银行客户，我希望收到电子形式的信用卡对账单。”“作为照片网站用户，我希望在浏览器中裁切和编辑照片。”

在敏捷开发中，测试和集成中也回避瀑布方法。单元和集成测试随着新功能代码创建，测试在构建过程中自动应用。代码发行中的唯一文档往往是提供初始需求的用户故事，和瀑布方法的功能性规格和需求文档形成强烈对比。用户故事来自于产品负责人维护的、排定优先级的积压功能故事。通过保持可以随着业务需求改变的优先级列表，维持开发的敏捷性，开发工作很容易对业务需求的变化做出反应，不会浪费精力和出现返工现象。

DevOps 是敏捷方法在系统管理上的应用。

8.6.2 什么是持续交付

持续交付（Continuous Delivery, CD）是软件交付和持续更新的一组原则和实践。软件交付指的是软件从源代码到随时可用的安装包的过程。这一过程包括软件包的构建（编译）和所有质量测试。如果构建或者测试失败，该过程停止。CD 起先是极限编程中的一种设计模式，但是之后成了单独的学科。

例如，实践 CD 的组织频繁构建软件，往往在检测到新的源代码更改之后立即进行。

更改触发构建过程，构建成功完成时触发自动化的测试过程。测试成功完成时，软件包就可以使用。

CD 不同于新软件版本不频繁发行（可能每年只有一次）的传统软件方法论。在后一种方法中，当软件发行日期临近时，构建软件包，可能涉及人工成分很大的过程，包括人工干预、可能还有临时的过程。测试混合了自动化和可能需要几天才能完成的人工测试。如果发现任何问题，整个过程都得重来一遍。

持续交付强调自动化、打包和可重复性，采用分担责任的文化以获得好的结果。在持续交付中，我们承认过程的每一步都是最终交付的软件及更新的一部分，该过程在每一阶段都是会发生。

系统运营和维护可以视为持续交付的一种形式。运营任务是软件从存储库打包构建进入系统的过程的一部分。虽然这些原则和实践来自于软件开发，但是很容易将它们映射到运营任务，并增进效率和可靠性。

Humble 和 Farley 编撰了持续交付的 8 条原则（2010）。

持续交付的 8 条原则

- 1) 发行 / 部署的过程必须可重复和可靠。
- 2) 自动化任何工作。
- 3) 如果某件工作很困难或者很痛苦，更多地实践以改进和自动化。
- 4) 将所有工作置于源代码控制之下。
- 5) “完成”意味着“发行、在最终用户手中正常工作。”
- 6) 构建良好的品质。
- 7) 每个人对发行过程负有责任。
- 8) 持续改进。

虽然这些原则都是不言自明的，但是第 3 条原则值得展开讨论。对于容易出错、痛苦和困难的任务，我们的反应往往是想办法减少执行这些任务的次数。持续交付则更经常执行这些任务以改进技能（“实践造就完美”），修复过程中的问题，并自动化这些过程。

持续交付还有 4 种实践。

持续交付的 4 种实践

- 1) 只构建二进制代码一次。
- 2) 对每种环境的部署使用相同的可重复过程。
- 3) 在部署环境上（例如，包含诊断）进行基本的功能测试（“冒烟测试”）。
- 4) 如果出现任何失败，停止过程并重新开始。

对于任何给定的发行周期，二进制包只构建一次。这与 QA 构建软件包并测试，然后由部署团队检查相同代码并构建软件包用于部署的环境形成了对比。后一种工作不仅有重复劳动，而且可能引入错误。多次构建请求可能引起混乱或者沟通不利，造成软件包每次

都从略有不同的源代码版本中构建。某个团队可能有兴趣“悄悄地进行一些小修复”，这似乎有益，但是很危险，因为这意味着新代码没有接受完整的端到端测试。验证第二次构建软件包时使用完全相同的 OS 发行版本、编译器发行版本和构建环境，是很困难的。

版本控制的构建

在 Tom 的某个前雇主那里，构建系统检查 OS 内核的 MD5 散列、编译器和其他一些文件。构建工具团队决定用某个特定的工具链构建 x.y 版本，构建系统不会让你用其他任何工具链构建该版本。如果客户 5 年以后要求对版本 X 打补丁，该公司知道可以精确地按照需要构建该软件，只有补丁是不同的项目，而编译程序、OS 或者其他工具都保持不变。这一点特别重要，因为该公司制作设计芯片的软件，一旦设计开始，芯片设计人员就不会改变工具链。如果你用版本 X 开始设计，供应商承诺只在芯片设计完成之后才修复特定版本中的缺陷。

通过尽可能多地自动化，为无法自动化的步骤创建文档记入的过程，我们创建了可重复的部署过程。在传统软件开发方法论中，发行工程师手工为 QA 环境构建二进制代码，并以临时方式部署。在持续部署中，构建是自动化的，每个阶段使用相同的部署过程。

“冒烟测试”是一种基本功能测试，就像插入一个设备，看看它会不会开始冒烟（这是不应该出现的！）。将内建的诊断测试作为构建和部署自动化的一部分，使你可以确信刚刚部署的软件符合基本功能性要求。如果内建测试失败，构建或者部署将输出错误，让你知道存在某种问题。

持续交付不在出现故障的时候打补丁，而是要求我们找出根源、修复问题。然后，我们重复该过程，验证问题是否已经消失。

CD 生成一个安装包，它经过充分测试，可以部署到生产环境。但是，实际投产该发行版本是一个业务决策。部署通常较不频繁，可能每天或者每周一次。部署团队可能执行更多的测试，实际部署本身可能很复杂，需要人工进行。自动化这些实践并频繁进行（可能和新软件包的出现一样频繁）组成了持续部署，这种方法将在 11.10 节进一步讨论。

8.7 小结

在本章中，我们解释了 DevOps 的文化和原则，并考察了其历史沿革——也就是敏捷和持续交付。本章中的原则和实践可以作为运营和维护任务中文化及态度转变的坚实基础。这种转变带来了可计量的效率，在有力的应用下可以增加正常运行时间，应该自行探索。

Theo Schlossnagle (2011) 将 DevOps 描述为对“世界运营主义”的自然反应。因为每类公司都在加快速度，最成功的竞争者是“专注于”每个业务单位运营的组织。

DevOps 不与某种技术相关，而与业务问题的解决相关 (Edwards 2010)。它始于对业务需求的理解和优化过程以更好地解决这些问题。

DevOps 使企业的运行更加顺畅，使涉及的人员更高效地协作。它给我们带来了在任何规模上顺利运营的希望。

练习

1. DevOps 有哪些基本原则？
2. 瀑布环境与 DevOps 环境有何不同？
3. 为什么 DevOps 不是一个职衔？
4. 描述敏捷和 DevOps 实践之间的关系。
5. 在最低限度上，谁的参与是 DevOps 协作的关键？
6. 指出 3 种发行工程最佳实践。
7. DevOps 转换过程涉及哪些基本步骤？
8. 描述你的发行过程（代码提交到生产环境中的运行）。识别人工步骤、团队间移交和定义不良或者不完整的步骤。根据 DevOps 的 3 条道路，可以进行哪些改进？

第9章 Chapter 9

服务交付：构建阶段

他们当然没有显现任何放慢的迹象。

——Willy Wonka

服务交付是创建服务的技术性过程，从开发人员创建源代码开始，结束于服务在生产环境中运行。完成所有这些工作的系统称作**服务交付平台**（Service Delivery Platform）。服务交付包括两个阶段：构建和部署。**构建阶段**（Build Phase）始于软件源代码，产生可安装的软件包。本章介绍这一阶段，**部署阶段**（Deployment Phase）取得这些软件包，准备服务基础设施（机器、网络、存储等），并产生运行系统。部署阶段在第10章中介绍。

服务交付流程和装配线类似，每一步完成某项工作。途中进行各种测试，验证产品正确工作，然后传递到下一步。缺陷得到检测和监控。

整个过程的运行越快，测试结果就越快得到。但是，全面的测试花费很长时间。因此，早期的测试应该是最快、最广泛的测试。随着信心的建立，进行更慢、更详细的测试。这样，最可能发生的故障出现在早期，在其他测试中可以消除该需求。例如，早期阶段编译软件，对容易检测的语法错误很敏感，但是对软件正常工作提供不了很多保证。性能、安全性和易用性的测试最为费时，人工测试留到最后，此时其他故障都已经检测出来，可以节约人工。

必知术语

创新：做我们之前未做过的（好）事。

利益相关方：在项目的成功上有利益关系的人或者组织。

工件：软件开发期间产生的任何有形的副产品——源文件、可执行程序、文档、用

例、框图、软件包等。

服务交付流程：系统自始至终的历程，往往简写为**流程（Flow）**。

周期时间：流程完成的频率。

部署：发行版本投产的过程。

门限：阻止流程的有意限制。例如，部署步骤的门限为质量保证测试是否成功。

发行候选：构建阶段的最终结果。不是所有发行候选都会被部署。

发行：整个流程（包括部署）成功完成。可能造成用户可见的变化。

好的服务交付平台考虑服务不仅是软件、还包括服务运行的基础设施这一事实。这包含了加载并正确配置操作系统的机器、安装和配置的软件包以及网络、存储和其他可用资源。虽然基础设施可以人工设置，但是最好自动化该过程。虚拟化实现了这类自动化，因为虚拟机可以通过软件操纵，软件定义的网络也可以。我们将越多基础设施当成代码处理，就能从软件开发技术（如版本控制和测试）中得到越多的益处。这种自动化应该视同于任何软件产品，并和任何应用程序一样经过相同的服务交付流程。

服务交付处理得当时，能够提供信心、速度和持续改进。应用程序和虚拟基础设施的构建、测试和部署可以完全自动化的方式完成，这一过程合理、一致、高效。这些工作也可以通过人工和临时的过程完成，但是那样无法保持一致，效率也较低。实现前一种状况是你的使命。

9.1 服务交付策略

有许多种可用的服务交付策略。大部分方法论归属于较旧的瀑布方法和较现代化的、与 DevOps 领域相关的方法。我们建议使用后者，因为它能得到更好的结果，促进创新更快出现。这种方法专注于自动化、测量和基于数据的改进。

9.1.1 模式：现代化的 DevOps 方法论

DevOps 方法论将平台分为两个阶段：构建阶段和部署阶段。部署阶段关心源代码的取得和安装包的制作。部署阶段获得这些软件包，在所要运行的环境中安装。每一步都执行测试，如果任何测试失败，过程将停止。源代码经过修订，过程重新开始。

图 9.1 展示了服务交付平台，其中有两个主要流程。上方的流程交付应用程序。下方的流程交付基础设施，4 个象限代表每个流程的构建和部署阶段。

每个阶段都有一个存储库：构建阶段使用源存储库。部署阶段使用包存储库。每个阶段都有一个控制台，提供对情况的可见性。应用程序流程和基础设施流程有相同的步骤，应该尽可能使用相同的工具。

部署在至少两个环境中的一个完成：测试环境和实际生产环境。服务最初在测试环境

中创建，该环境没有暴露给客户。在这个环境中，对发行版本进行一系列测试。当发行版本通过这些测试，就会成为发行候选。发行候选是可能成为最终产品的版本——部署在生产环境供客户使用的版本。

软件可以部署在其他环境，如专用沙箱（Private Sand-box）环境，工程团队可以在这种环境中进行测试环境中无法进行的破坏性试验。单独的开发人员应该每人至少有一个专用环境，用于自己的开发需求。这些环境通常是较大系统的精简版本，甚至运行在开发人员自己的便携电脑上。还可能有利于其他目的的不同环境。例如，可能有一个演示环境，用于利益相关方预览新发行版本。另外，新发行版本可能部署在一个中间生产环境，在部署到主生产环境之前供“早期访问客户”使用。

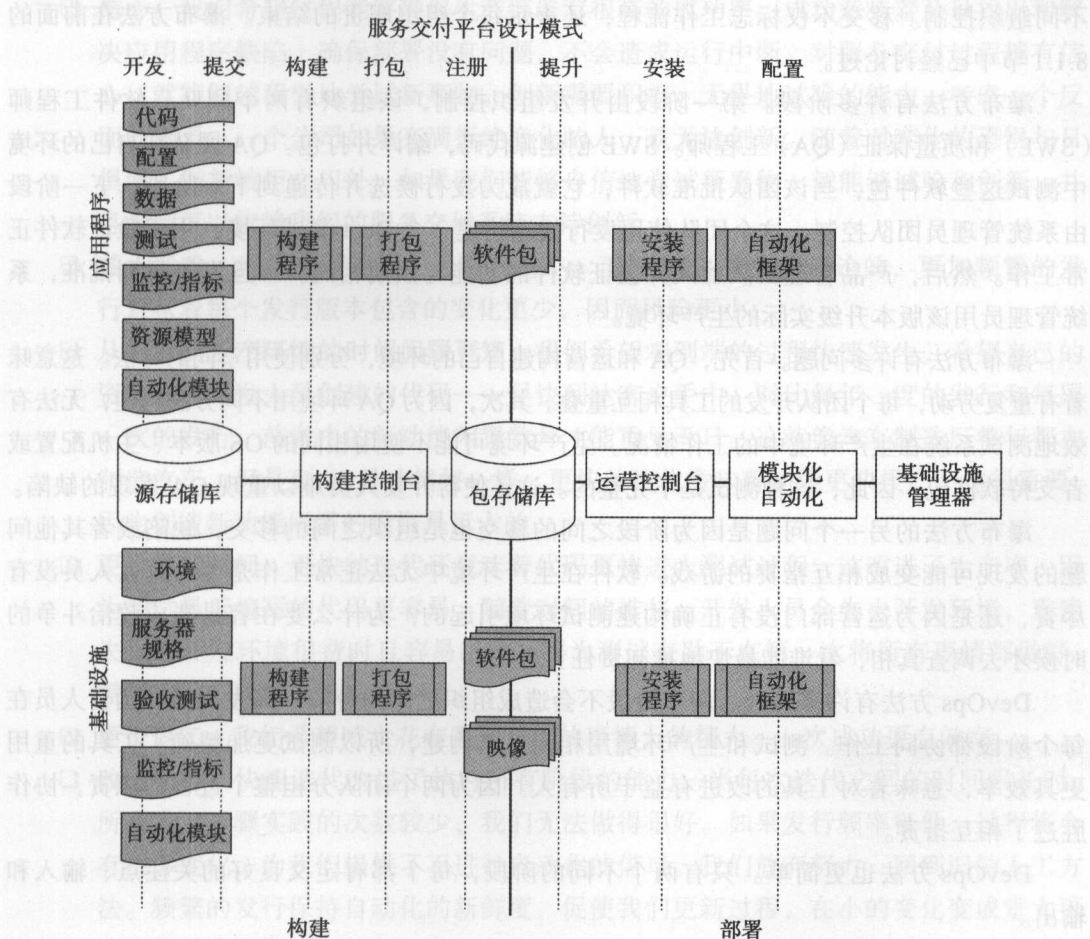


图 9.1 现代服务交付平台模式的各个部分（在 DTO Solutions 的 Damon Edwards 许可下重印）

最低限度，每个站点必须有单独的测试和生产环境，测试环境要和生产环境完全相同。开发人员自己的环境通常控制得没有那么好，在这种环境中测试可能遗漏某些问题。将发

行版本从开发人员测试直接移入生产环境，或者没有用于测试配置管理系统的环境，都是一种疏忽。没有理由不投资附加的基础设施，以建立合适的测试环境——这样做总能提高服务可用性，证明投资的价值。

虽然人们可能将构建阶段视为开发人员的领域，部署阶段视为运营人员的领域，但是在 DevOps 环境中并非如此。两个小组分担构建和使用整个系统的职责。各个步骤之间的移交标志着工作流程，而非组织边界。

9.1.2 反模式：瀑布方法论

瀑布方法的工作方式与现代化的 DevOps 方法不同。它以多个阶段为基础，每个阶段由不同组织控制。移交不仅标志工作流程，还表示每个组织职责的结束。瀑布方法在前面的 8.1.1 节中已经讨论过。

瀑布方法有许多阶段。第一阶段由开发组织控制，该组织有两个团队：软件工程师（SWE）和质量保证（QA）工程师。SWE 创建源代码，编译并打包。QA 团队在自己的环境中测试这些软件包，当该团队批准软件，它就成为发行候选并传递到下一阶段。下一阶段由系统管理员团队控制，这个团队使用发行候选构建一个 Beta 测试环境，用于验证软件正常工作。然后，产品管理团队加入并验证软件的功能与预期相同。一旦发行得到批准，系统管理员用该版本升级实际的生产环境。

瀑布方法有许多问题。首先，QA 和运营构建自己的环境，分别使用不同的方法。这意味着有重复劳动，每个团队开发的工具相互重叠。其次，因为 QA 环境用不同方法构建，无法有效地测试系统在生产环境中的工作情况。生产环境可能不使用相同的 OS 版本、主机配置或者支持软件包。因此，这种测试是不完整的。这还使得开发人员难以重现 QA 发现的缺陷。

瀑布方法的另一个问题是因为阶段之间的移交也是组织之间的移交，缺陷或者其他问题的发现可能变成相互指责的游戏。软件在生产环境中无法正常工作是因为开发人员没有尽责，还是因为运营部门没有正确构建测试环境引起的？为什么要在容易造成政治斗争的时候才去调查真相，看谁能最快地推卸责任？

DevOps 方法有许多好处。两个阶段不会造成组织之间的分隔，开发人员和运营人员在每个阶段都协同工作。测试和生产环境用相同工具构建，所以测试更加精确。工具的重用更具效率，意味着对工具的改进有益于所有人。因为两个团队分担整个过程的职责，协作胜过了相互指责。

DevOps 方法也更简单。只有两个不同的阶段，每个都有定义良好的关注点、输入和输出。

9.2 高质量的良性循环

好的流程造就了高质量的良性循环。严格的测试建立了稳固的基础，产生更好的发行

版本。这增强了信心，从而促进更快、更好的发行。因为发行版本较小，测试也得到了改进，然后，重复这一循环。

讨论服务交付时，人们往往专注于发行周期的速度。当某人炫耀自己将周期时间从6周缩短为1个小时，他们实际上忽略了重点。

真正重要的是对最终产品质量的信心。改善代码管理、速度、打包和周期时间都是手段，而非目的。平台提供更好的测试过程以及其他确保一致性的自动化过程的能力，是信心的来源。在 DevOps Cafe 播客的第33集中可以听到一个出色的讨论（Willis、Edwards & Humble 2012）。

更确切地说，好的服务交付平台应该得到如下的结果：

- ❑ **信心**：我们希望过程能够确保生产部署有很高的成功率。成功意味着及早找出和解决应用程序缺陷，确保部署没有问题，不会造成运行中断。对服务交付过程越有信心，就能够越积极地尝试新事物。创新需要积极、无畏地试验的能力。考虑一个反面的例子：一个公司如果充满拒绝变化的人，就无法创新。随着对变化的恐惧与日俱增，创新被拒之门外。如果我们能够自信地尝试新事物，就能够试验和创新，并且深信可以依赖我们的服务交付系统支持创新。

- ❑ **减小风险**：迭代速度越快，风险就越小。正如8.2.4节中所讨论的，更加频繁的发行意味着每个发行版本包含的变化更少，因而风险更小。

- ❑ **从键盘到生产环境的时间间隔更短**：我们希望端到端的过程快速发生，希望自己的资本——开发人员创建的代码——尽快到达客户手中。对比每年一度的发行和每周一次的发行，前者中的新功能闲置数月才能重见天日。这就像汽车制造厂整年都在制造汽车，但是到12月才销售一样。更快的迭代意味着功能更快投产。这很重要，因为创造新功能所需的投资是巨大的。

- ❑ **更少等待时间**：更快的迭代还意味着代码更快进入测试过程。这改进了生产率，因为调试最近编写的代码更容易。随着时间的推移，开发人员会失去开发环境，重建失去的开发环境很费时且容易出错。因为测试有助于在第一次将所有事情都做好，需要重做一遍的工作就更少了。

- ❑ **更少返工**：我们希望减少花在重做过去的事情上的精力。一次成功更有效率。

- ❑ **改进执行**：快速迭代改进了执行所有阶段的能力。当每次迭代之间的时间很长时，所有人工步骤实践的次数较少，我们无法做得很好。如果发行频率极低，过程将会有很多变化，为我们提供了不进行自动化的借口。我们放弃努力，回到旧的人工方法。频繁的发行保持自动化的新鲜度，促使我们更新过程，在小的变化变成重大更改之前反映它们。

- ❑ **持续改进的文化**：理想的过程总在发展和改进。最初，它可能包含人工步骤，这是意料之中的，因为过程刚发明出来时具有可塑性。一旦端到端的过程自动化，就可以进行测量并自动收集指标。利用这些指标可以进行数据驱动改进。对于持续改进

的过程，我们不仅需要合适的技术，还需要接受变化的文化。

□ **改善的工作满意度：**看到我们的更改快速投产，是令人激动和鼓舞人心的。当工作和得到回报的间隔很短时，我们就会将二者关联起来。因为工作立刻给我们带来喜悦，对工作的满意度也就得到了改善。

团队不仅要关注周期时间，还应该有平衡软件交付平台各方面速度的指标。我们建议每个 DevOps 团队收集如下指标：

1) **缺陷解决时间：**从初始缺陷报告到修复的代码生产部署的时间。

2) **代码提前期：**从代码提交到生产部署的时间。

3) **补丁提前期：**从供应商补丁发行到生产部署的时间。

4) **部署频率：**每月完成的生产部署次数。

5) **服务平均恢复时间：**运行中断持续时间，从最初发现到恢复服务的时间。

6) **更改成功率：**成功生产部署次数与总生产部署次数的比率。

9.3 构建阶段的步骤

构建阶段目标是创建部署阶段使用的安装包，它有 5 个步骤：

1) 开发代码。

2) 代码提交到源存储库。

3) 构建代码。

4) 打包构建结果。

5) 注册软件包。

每一步都包含某种测试。例如，软件构建验证编译，打包则验证所有必要的文件可用。如图 9.1 所示，源存储库在这一阶段作为主要的存储设施。最终输出存放在包存储库，移交给下一阶段。

9.3.1 开发

在开发 (Develop) 步骤期间，工程师编写代码或者制作其他文件。例如，他们可能编写 C++、Python 或 JavaScript 代码。图形设计人员创建图像和其他工件。

工程师们从源存储库中访问现有文件，将源代码下载到工程师的机器或者工作空间，在那里编辑、修订和更改文件，创建新文件。例如，致力于某项新功能的软件工程师可能对所有相关文件进行许多修订，编译和运行代码，并重复这一过程直到源代码按照预期编译和运作。

9.3.2 提交

在提交 (Commit) 步骤中，开发的文件上传到源存储库，这一般不会频繁进行，因为

这表示文件已经达到某种完整性水平。所有提交的代码应该是可工作的代码，否则，其他开发人员的工作将暂停。他们将把最近的更改放入自己的工作空间，结果将是无法工作的代码。开发人员无法得知问题是出在自身的错误，还是源代码的质量不佳。签入无法工作的代码被称作“打断构建”。构建工作可能因为这些代码不再编译或者代码编译但是自动测试（后面将做讨论）失败而中断。如果构建被打断，返回到可工作状态应该是高优先级的任务。

这一步的门限是**提交前检查**（Pre-submit Check）。为了避免明显不好或者损坏的代码进入，源存储库系统可以配置为调用程序检查新文件，进行基本校验，拒绝提交失效代码的企图。这些检查不能确定代码是否完善且没有缺陷，但是可以检测明显的错误，如语法错误。这里常常运行后面描述的单元测试，以验证更改不会破坏基本功能。提交前检查常常检查风格指导方针的一致性（在 12.7.4 节中讨论）。

因为提交前检查可以调用任何程序，人们已经发现了许多创造性地运用它们的方法，超越了简单的完备性检查。提交前检查可以用于实施策略，更新状态显示和检查常见的缺陷。例如，我们曾经遇到权限设置不正确的一个文件导致的运行中断。现在，采用了提交前检查预防相同问题再次发生。

9.3.3 构建

在**构建**（Build）步骤期间，处理源文件以生成新工件。这通常意味着源代码经过编译生成可执行文件。这一步可能还要执行其他任务，如将图像从一种格式转换为另一种，从源代码中提取文档，运行单元测试等。

这一步根据所有构建过程是否成功完成而设置门限。这些检查中最重要的是**单元测试**（Unit Test）。单元测试是可以在编译单元（如函数库和对象类定义）上运行的质量保证测试。相比之下，下一章讨论的**系统测试**（System Test）包括运行服务和测试其全部功能。

单元测试往往采取附加可执行文件的形式，除了以不同方式调用代码中的函数，检查结果是否符合预期之外，该文件不做任何事情。例如，假定源代码包含操纵用户名的函数库。假定库中的一个函数测试某个字符串是否能够作为有效的用户名。单元测试可能多次调用该函数，每次测试一个已知无效、形式不同（太短、太长、包含空格、包含无效字符）的字符串，验证所有情况下用户名是否都会被拒绝。另一个单元测试可能反转已知有效的字符串。

单元测试可能相当复杂。例如，为了测试需要访问数据库的函数，单元测试代码可能用样板数据建立一个小型数据库。建立一个数据库很复杂，所以已经开发了测试框架，允许人们为测试目的替换函数。例如，假定你要测试打开数据库连接、发送查询和操纵结果的函数。你可以用一个不做任何操作的函数代替“连接到数据库”函数，用总是返回特定结果集的函数替换“查询数据库”函数。现在，可以测试函数而无须拥有真正的数据库。

这一步为我们提供了在过程早期执行主动测试的机会，避免在以后浪费精力。

9.3.4 打包

在打包 (Package) 步骤期间, 从前一步留下的文件用于创建安装包。包是一个文件, 包含所有待安装文件的编码, 以及执行安装的机器可读指令。因为是单一文件, 更容易传输。

这一步的门限根据包创建是否成功设置。简单的包格式可以是包含将要安装的所有文件的 Zip 或者 tar 文件, 加上一个安装脚本。运行安装程序时, 它读取包, 解压所有文件, 然后运行安装脚本。更详细的描述可以在《The Practice of System and Network Administration》第3版 (Limoncelli, Hogan & Chalup 2015) 中的“软件存储库”一章中找到。

软件包应该设计为在任何环境中运行。不要为测试环境和生产环境创建单独的包, 更不要为测试构建软件包, 在测试之后重新构建用于生产环境。生产环境应该运行经过测试的包, 而不是类似于测试环境中使用的包。

什么是软件包?

软件包是一个容器。这个文件包含安装应用程序、补丁或者程序库所需的一切内容。包通常包含所要安装的二进制可执行文件、相关的所有数据文件、配置数据和描述软件安装和删除方法的机器可解读指令。你可能熟悉 Zip、Unix tar 和 cpio 等文件格式。这些文件包含许多较小文件内容和元数据。元数据就像目录或者索引, 它编码解压单独文件和文件所有权、权限和时间戳所需的信息。

9.3.5 注册

在注册 (Register) 步骤期间, 包被上传到包存储库。此时包已经为移交给部署阶段做好了准备。这一步根据上传是否成功设置门限。

9.4 构建控制台

构建控制台 (Build Console) 是管理所有构建步骤的软件, 它方便了结果和过去历史的查看, 保存有关成功率、过程所花费时间等统计值。构建控制台总是基于 Web 的工具, 提供查看状态的仪表盘和管理过程的控制面板。有许多此类工具, 包括 Hudson、Jenkins CI、TeamCity、Go Continuous Delivery 和 Atlassian Bamboo。

一旦找到喜欢的工具, 你会发现自己希望将其用于所有地方。因此, 选择此类工具时要确保它能和当前使用的工具 (源代码存储库软件、编译器和其他构建工具) 一起使用, 并且提供对你所使用的所有操作系统和平台的支持。这类工具通常可以通过某种插件机制扩展。这样, 你就没必要寄希望于供应商进行扩展了。如果有一个开源开发人员社区维护免费的插件, 就是一个好的迹象, 这通常意味着该工具可以扩展, 且得到很好的维护。

构建控制台还应该有一个用于控制它的 API。你应该可以编写工具与其交互、启动任

务、查询等。最简单和有用的 API 之一是最近完成的构建的一个 RSS feed。其他的许多系统可以读取 RSS feed。

案例研究：构建状态的 RSS Feed

StackExchange 是一个内部聊天室系统，有能力监控 RSS feed 并在指定的聊天室中通知任何新条目。SRE 聊天室监控构建完成的 RSS feed。每当构建完成，会有一个包含构建内容和成功与否的通知，以及指向状态页面的链接。这样，整个团队都能看到他们的构建情况。

9.5 持续集成

持续集成 (Continuous Integration, CI) 是在一天中自动多次进行构建阶段工作的方法。构建阶段的每一轮运行由某种事件 (通常是代码提交) 触发。然后，所有构建阶段步骤以全自动化方式运行。

所有构建都从源代码存储库的主干进行。所有开发人员直接向主干投送代码，不为未来的开发创建长期分支或者独立工作区。

“持续”一词指的是每次更改都得到测试这一事实。想象一个描绘待测试更改的图表。在 CI 中，这条直线是完整的，也就是连续的。在其他方法中，并不是每个发行版本都经过测试，直线应该是不完整或者不连续的。

自动触发构建过程而不通过人工触发的好处不仅仅是不需要有人等着启动该过程。因为过程在每次代码提交时都触发，因此很快就会发现错误和不好的代码，问题也就更容易找到，这是因为 8.2.4 节中讨论的“小批次”原则，因为更改的环境仍然在开发人员的短期记忆中，问题也就更容易修复。而且，在有缺陷的代码上进行新更改的可能性也较小，这种更改可能需要逆向工程或者重新设计，浪费每个人的时间。

这么频繁地执行过程，还能确保其保持自动化。过程中破坏系统的小更改可以在影响较小的时候就得到修复。对比每月一次的构建：到下一轮构建时，过程中的大部分可能已经变化。破坏可能很大，恢复人工过程的诱惑变成了真正的问题。

不要自行注册软件包

只有通过控制台构建的包才允许在包存储库中注册。换言之，人们不能注册软件包。开发人员通常有从自己的工作站运行整个构建阶段，对其进行维护和调试的手段。但是，他们能够创建软件包，并不意味着应该允许他们将软件包上传到存储库。他们构建的软件包可能有意无意地依赖他们的工作站或者环境。通过控制台构建所有包，可以确保任何人都能构建软件包。因此，好的规则是：正式的软件包总是由构建控制台自动化措施构建。

从人工构建转向全自动构建可能非常困难，有 3 个关键成分。首先，确保可以从头到尾人工执行该过程。如果构建不经常进行、不同人执行过程的方式不同或者由不同的人执行不同步骤，这一点可能成为挑战。其次，确保源文件都保存在一个源代码控制系统中。如果构建过程中的一步是向平面设计人员询问是否有更新的图像，那么过程就不完整。平面设计人员有更新的图像时应该可以签入到存储库中。构建系统应该简单地获得最新提交的文件。第三个成分是自动化每个步骤，直到不需要交互式（键盘 / 鼠标）输入。在那个时候，所有步骤可以加载到控制台并调试，最后成为新的正式过程。

CI 似乎是显而易见的事情，但是你可能会为许多主流公司的构建工作很不频繁而感到吃惊。

高风险的 14 天构建

旧金山的一家大部分系统管理员熟知的软件公司采用一种构建过程，该过程是人工进行的，需要两周才能完成。其软件发行发生在每个季度的第一天。两周之前，一位工程师开始构建。如果开发人员的工作延误，构建使用的时间更少。软件需要为 3 种 Windows 版本和十多种 Linux/Unix 变种构建，有些构建步骤很不可靠且是临时性的，使整个过程处于危险之中。构建过程中发现的缺陷将对特定 OS 进行“热补”，以避免因为返工而危及时间安排。每个季度，该公司都处于推迟交付的危险之中，这种情况暴露在众目睽睽之下，令人尴尬。

一位新雇用的构建工程师对这一过程感到震惊，他知会首席技术官（CTO），说明自动化应该是头等大事。CTO 不同意这一观点，并解释过程不需要自动化的理由：“我雇佣你就是做这个的！”

这位工程师不顾一切地进行自动化。在几个月内，所有操作系统所用的整个过程都完成了自动化。之后，该过程被放入构建控制台，实现 CI。

这一举措改变了开发组织。开发人员现在更有效率，制作出了更好的软件。发行工程师可以将焦点放在更重要、更有趣的工作上。

在这种情况下，反抗 CTO 是正确的举动，这位构建工程师是英雄。

9.6 以软件包作为移交接口

在转向部署阶段之前，我们需要暂停一下，讨论两个阶段之间发生的移交。

构建阶段和部署阶段之间的移交涉及安装包的交付。使用软件包更容易确保在测试和部署中使用相同的代码。部署可能发生在测试之后数小时或者数周，在其他部署中使用相同文件很重要。否则，可能引入未经测试的更改。

管理包比管理单独文件更容易。包通常使用散列方法加密或者数字签名，接收方可以验证该文件在途中没有被修改。上传单个文件也比上传文件的层次结构更安全。因为元数

据被编码到包中，不用担心在此期间不小心被更改。

应该避免使用在各阶段之间传递文件的其他机制。我们曾经看到在团队之间用电子邮件传送单独文件，丢失了授权和所有权元数据。我们还曾经看到许多组织将所有单独文件放在文件服务器上的一个特殊子目录中进行移交。无法用任何简单的方式得知测试和部署之间这些文件是否已经更改。维护该目录的人在我们完成当前文件之前无法准备下一次发行。在这种系统中，无法访问旧的发行版本，更糟糕的是我们发现这一过程涉及某人主目录下的一个子目录，这意味着如果那个人离开公司，整个过程将被破坏。

还要避免用源代码存储库作为移交机制，源存储库通常有在特定时刻标记所有文件的功能。然后，将标记名称交给部署阶段，作为移交手段。这种方法可能造成许多问题。例如，难以验证所有文件和元数据的完整性，授权可能不小心改变，文件所有权会被有意改变。如果采用这种技术，将二进制代码签入存储库，存储库将明显增大、无法控制。如果不签入二进制代码，就意味着部署人员不得不构建它们。这是重复劳动，并且有引入绕过测试的微小变化的风险。

9.7 小结

本章概述了服务交付，并详细解释了构建阶段。下一章将详细解释部署阶段。服务交付由技术过程组成，包括将源代码转化为运行的服务。服务交付平台是驱动该过程的软件和自动化措施。

服务交付有两个阶段：构建和部署。构建将源代码变成软件包，部署取得软件包并将其部署到某个环境。不同目的有不同的环境。测试环境用于测试服务，实际环境是为客户提供服务的运行环境。这两个环境应该以相同方式设计和构建，使测试尽可能有意义。测试环境通常仅在规模和存储的是虚构数据而非用户数据这两方面上不同。还有许多其他环境，包括用于探索性测试、性能测试、Beta 测试等的环境。

流程是整个服务交付平台的一次历程。每个步骤包含旨在尽早找出缺陷的测试。实际环境中找到的缺陷是早期阶段和环境中没有进行充分测试的结果。

当服务交付平台运作在最佳状态时，结果是对交付服务的高度自信。因此，组织可以更积极地进行更改，各种功能也更快发布。换言之，创新得到加速，这和对交付系统没有自信的组织形成了对比。在后一种情况下，发行版本需要几个月制作，缺陷在编码编写很久以后才发现，更正问题更加困难。由于恐惧和令人失望的规则，更改被扼杀了，从而阻碍了创新。

从构建到部署移交的是一个软件包——包含许多文件的文件。比起传输许多单独文件，包更容易传输、更安全、也更不容易出错。

例如，假定构建一个环境需要下载软件包A、B和C。构建阶段的每一步产生一个新的

练习

1. 描述服务交付平台构建阶段的各个步骤。
2. 为什么构建和部署阶段是分离的？
3. 为什么应用程序和基础设施流程是分离的？
4. 瀑布方法和现代化的 DevOps 方法相比有何优劣之处？
5. “好流程”如何促使企业实现其目标？
6. 描述持续集成及其好处。
7. 为什么测试和生产需要不同环境？
8. 描述你的组织所使用的服务交付平台。根据本章描述的好处，你建议做哪些改变？
9. 使用软件包作为移交机制有何好处？
10. 你的组织是否使用 CI？你当前平台的哪些部分需要更改才能实现 CI？

服务交付：部署阶段

全速前进！

——Elvia Allman 《致 Lucy 和 Ethel》

在前一章中，我们解释了构建阶段，该阶段结束于软件包的创建。本章我们将解释部署阶段，使用软件包创建运行服务。

部署阶段在一个或者多个测试及生产环境中创建服务。决定用于测试环境的发行版本是否为生产环境中的使用做好准备需要批准。

部署阶段的目标是创建一个运行环境。然后使用这个环境测试或者运行实际的生产服务。

如图 9.1 所示，软件包从包存储库读取，然后安装和配置，以创建一个环境。创建的环境可能是测试环境，用于验证系统所有部件是否能够一起工作，也可能是实际环境，用于向用户提供服务。另外，还可能是之前 9.1.1 节中描述的其他环境。

10.1 部署阶段的步骤

部署阶段有 3 个步骤：软件包升级、安装和配置。

10.1.1 升级

升级（Promotion）步骤是某个发行版本被选中并升级到预期的环境中使用。被选中并标记为某环境合适版本的预期版本进行构建。

例如，假定构建一个环境需要 3 个软件包 A、B 和 C。构建阶段的每一遍产生一个新的

软件包。包 A 有 1.1、1.2 和 1.3 版本，B 有 1.1、1.2、1.3 和 1.4 版本。版本较多是因为签入的次数更多。C 有 1.1 和 1.3 版本（1.2 版本缺失是因为出现了构建错误）。

假定我们已经测试了 A-1.2、B-1.4 和 C-1.3 的组合，并得到投产批准。升级步骤将告知包存储器，将其标记为指定的生产版本。

正如 9.1.1 节所述，上述的特定版本选择通常需要在生产、beta 和早期访问环境中进行。但是，开发和测试环境可能简单地使用最新发行版本：A-1.3、B-1.4 和 C-1.3。

特定环境下软件包的标记方式取决于包存储库系统。有些使用标签机制，例如在某个时刻，只有特定软件包的一个版本可以有“生产”标签。可能有许多软件包，每个都指定一个版本为其生产版本。通常，有一个称作“最新”的虚拟标签，自动指向最新版本。

有些存储库系统使用“图钉”（Pinning）技术。软件包被固定在特定的版本上，即使有更新的版本可用，也总是使用该版本。标签通常是全局的，而图钉在环境级别上使用。例如，测试和实际环境各自将软件包固定在不同版本上。

其他包存储库系统的工作方式完全不同，它们在某一时刻只能存储包的一个版本。在这种情况下，将有多组存储库，包可以在它们之间复制。例如，所有新软件包放在称作“开发”的存储库中。当包为测试环境使用做好了准备，它被复制到“测试”存储库中。测试环境中的所有机器指向该存储库。如果包被批准用于生产，它被复制到第三个存储库——“生产”存储库中，生产环境中的所有机器从该存储库读取包。这类系统的优点是生产环境不会不小心安装未经批准的包，因为它不知道这些软件包的存在。但是，保持过去软件包版本的历史更加困难，有时候需要某种单独的存档子系统。

10.1.2 安装

在安装（Installation）步骤中，软件包被复制到机器上并安装，这由理解包格式的安装程序完成。大部分操作系统都有自己的安装程序软件，每个软件一般都绑定到原生的包存储库系统。

包可能包含安装之前和之后运行的脚本。实际安装过程包括运行安装前脚本、从包中将文件复制到最终位置，然后运行安装后脚本。安装前脚本完成创建目录、设置权限、验证先决条件是否满足、创建拥有所安装文件的账户和组等任务。安装后脚本完成在默认配置不存在时复制默认配置、启用服务、将安装注册到资产管理器等任务。安装后脚本还可以执行冒烟测试，如验证访问远程服务、中间件和数据库等其他基础设施服务的能力。

10.1.3 配置

在配置（Configuration）步骤中，准备本地设置和数据，将已安装软件包变成运行的服务。

软件包常常包含进行一些通用配置的安装脚本，但是这一步完成创建工作服务所需的机器特定工作。例如，安装一个 Web 服务器软件包，会创建一个通用 Web 服务器，配置为托管来自特定目录的静态文件。但是，决定从该机器提供服务的域、让负载均衡器知道它

的存在以及特定于机器的其他任务将在这一步中完成。

配置步骤的门限由健康检查设置——验证系统运行的几个简单测试。例如，常见的健康检查通过请求仅在执行几个快速内部测试之后才响应的特定 URL 进行。

有许多用于配置管理的软件框架，流行的框架包括 CFEngine、Puppet 和 Chef。它们都可以创建配置特定服务的模块，并在必要时将模块应用到不同的机器。配置管理在 12.6.4 节中进一步讨论。

配置的两种主流策略是收敛编排（Convergent Orchestration）和直接编排（Direct Orchestration）。收敛编排是环境配置方式的一个描述，然后配置管理系统进行单独的更改，使整个系统收敛于预期的状态。如果因为某种原因进行了不应有的更改（用户无意或者有意，或者机器故障等外部事件引发），编排系统将检测出这种情况，并做出更改直到预期配置再次收敛。当下一次定义配置时，系统开始向新定义收敛，做出达到新状态所需的最少更改。收敛编排可以描述为使环境达到特定状态并保持。

直接编排可以描述为执行某种多步骤过程的一种方法，在此过程中某些约束条件一直成立。例如，将数据库从一台机器转移到另一台机器需要许多步骤，必须以某种顺序进行，在此过程中，“客户端总是可以访问到数据库”的约束条件一直成立。

下面是可能采取的步骤：

- 1) 机器 B 被配置为数据库副本。
- 2) 等待副本与主数据库同步。
- 3) 数据库客户端进入临时的只读模式。
- 4) 机器 A 和 B 的角色切换，使 A 变成只读副本。
- 5) 数据库客户端离开只读模式，配置为将写操作发送到机器 B。

用收敛编排实现这一目标需要一个难以处理的过程，必须为每一步创建预期状态，等待一个状态实现之后才能切换到下一步。

直接编排的一个挑战是如何处理同时发生的多个过程。例如，想象在系统中加入负载均衡器且 Web 服务器重新配置以添加新服务的同时发生这一过程。这些过程都涉及相互重叠的机器和资源集。上述步骤必须排定顺序和协调，以避免冲突，确保系统不会陷入绝境。目前这是人工完成的，是容易出错的过程。大规模地对这些步骤进行自动化，是研究人员刚刚开始考虑的。

转移到收敛编排的障碍之一是系统架构必须支持。这是自动化部署和基础设施管理过程的企业所遇到的主要问题之一，特别是无法修改的商业化产品。内部制作的系统可以从一开始就设计为支持收敛编排，也可以在事后修改。

10.2 测试和批准

在发行版本投产之前，必须经过测试和批准。首先，完成自动测试，接下来执行人工

测试（如果有的话）。最后，管理层批准或者同意发行。在发行时必须同意的一系列人或者部门称作批准链（Approval chain）。在这类活动全部完成之后，发行版本可以升级投产。

10.2.1 测试

测试包含许多不同的类别。在构建阶段，单元测试在每个组件上进行。部署阶段有 4 类测试：

- ❑ **系统测试**：这种测试组合服务的所有部件，测试最终产品或者系统。系统测试在测试环境中运行的服务上进行，通过这些测试是发行版本在生产环境或者包含外部客户的任何其他环境中使用的先决条件。每个单独的功能都应该进行测试。多步骤的工作流（如购物）也应该测试。有一些测试框架能够执行测试，就像由用户执行一样。例如，Selenium WebDriver 是一个自动化网站测试的开源项目，它发送 HTTP 请求，就像这些请求来自不同 Web 浏览器一样。不管用户如何与软件交互，都可以有自动化测试的工具。这些交互方式包括基于 PC 的 GUI、API、控制台/键盘、手机，根据 Gruver、Young & Fulghum（2012）的文档，甚至还包括激光打印机的面板。
- ❑ **性能测试**：这些测试确定服务在各种条件下的速度，在测试环境中或者特别构建的性能测试环境中的服务进行。它应该确定性能是否满足书面的规格或者需求。但是，这种规格往往并不存在或者很模糊。因此，性能测试的结果往往只能和以前的结果相比。如果整个系统或者特定功能比前一发行版本明显慢——**性能退化**（Performance Regression）——则测试失败。
- ❑ **负载测试**：这类特殊的性能测试确定系统能够维持的负载。负载测试通常在测试环境或者特殊的性能测试环境中进行。这类测试包括让服务承受越来越大的流量（或者负载），以确定系统能够处理的最大负载。举个例子，Google 首先进行负载测试，以验证 Linux 新内核中的更改对搜索群集承担的负载量没有负面影响，然后才投入使用。为此，Google 构建了一个运行该内核发行版本的群集。人为生成的搜索查询产生越来越大的 QPS，最终群集达到极限，无法达到更高的 QPS 值，或者系统变得很慢，无法在要求的毫秒数内应答查询。如果最大 QPS 明显低于前一个发行版本，Google 就不会升级到该内核。
- ❑ **用户验收测试**（User Acceptance Testing, UAT）：这种测试由客户进行，验证系统符合他们的需求，并验证制作者声称的能力。客户运行自己的测试，验证新发行版本满足其需求。例如，他们可能运行涉及服务的每个业务过程。系统测试让开发人员确保交运的产品没有缺陷，而 UAT 让客户确认收到的产品没有缺陷。理想情况下，为 UAT 开发的任何测试应该透露给开发者，以便加入他们的成套测试中。这样，客户的关注点就可以在过程的早期进行验证，遗憾的是，这一点并不总能做到。UAT 可能包含使用实际数据的测试，这些数据无法共享，如个人可辨识信息（Personally Identifiable Information, PII）。UAT 还可能用于确定内部过程是否需要修订。

每一步都有门限

StackExchange 的构建过程有几次不同的移交。每种都由一个测试作为门限，类比赛配线，就是不让不合格品流入下一道工序。测试有许多种，每种是一个单独的模块。服务交付流程有如下移交和门限：

- 1) 代码构建及之后以单元测试为门限的打包。
- 2) 数字签名验证以打包程序能否将代码传递给测试环境作为门限。
- 3) 将一个发行版本从测试环境升级到生产环境，以系统测试作为门限。
- 4) 生产环境升级成功以健康检查为门限。

10.2.2 批准

如果所有测试通过，发行版本称作**生产候选**（Production Candidate）。这些候选将通过一个批准过程，如果得到批准，将在生产环境中安装。

此时，批准链的成员们被要求就生产候选签署意见。批准链是必须对生产发行签署意见的一系列具体人员或者其代理人。例如，批准链可能包括产品经理、市场主管和工程主管。往往还需要安全、法律、隐私合规部门的批准。

每个环境都可能有不同的测试集和批准链，以限定可以进入的发行版本。在开发和测试环境中部署发行版本是自动批准的。UAT 和性能测试通常选择通过系统测试的最新发行版本。包含实际用户的环境（如 Beta 和演示环境）可能自动定期升级。例如，演示环境在每周的某个日子或者正式发行日之前的一段时期常常被清除并重新加载。

暴露给客户（或者产生收入）的环境应该经过最为严格的审查。因此，生产环境通常要求前述的所有测试，以及整个批准链的正面确认。

10.3 运营控制台

运营控制台（Operation Console）是管理运营过程（尤其是部署步骤）的软件。和构建控制台类似，这是一个基于 Web 的系统，方便查看结果和过去的历史，并保留有关成功率、过程持续时间等统计数字。

关于构建控制台的几乎所有说明都适用于运营控制台，所以可以参考 9.4 节。在此，安全和授权可能更为重要，因为过程可能影响到实际服务。例如，对谁可以启动新发行版本的投产可能有更严格的控制。

10.4 基础设施自动化策略

几个策略性技巧将帮助你全面自动化部署阶段，使其在控制台上无人值守运行。前面

已经讨论过，有一个用于部署基础设施的流程和另一个部署服务的流程。全栈部署可以进一步分解：准备和测试物理或者虚拟机器、安装操作系统、安装和配置服务。每一步是可以独立自动化的离散步骤。实际上，在大的环境中，你会发现每一步由不同团队负责。

10.4.1 准备物理机器

准备物理机器包括拆箱、安装到机架、连接电缆、配置 BIOS 设置和测试。前面几步需要体力劳动，非常难以自动化。只有极少数公司能够采用机器人完成这些步骤。但是，我们可以一次安装一个机架而不是一台机器，从而利用规模经济减少人工。改善这一过程的另一种方法是使用刀片服务器（Blade Server）。刀片服务器是由容纳许多单独计算机的一个机架组成的一种技术，每台计算机在一个“刀片”上，使得安装和维护更加容易。在超大规模环境中，机器被设计为具备特殊功能，实现快速、高效率的批量安装。Google 和其他公司设计自己的硬件，确保设计功能符合需求。

我们曾经看到过一些令人印象深刻的新硬件初始化过程自动化系统。Tumblr Invisible Touch 系统自动化固件升级、设置基板管理控制器（Baseband Management Controller, BMC）、将机器添加到 Tumblr 库存系统、执行几小时的压力测试和配置网络。

应急解决方案之一是人工安装硬件和配置 BIOS 设置，但是自动化设置正确性验证的过程。由于 BIOS 设置很少改变，这对于某些网站已经足够好，或者至少是完全自动化之前的折衷手段。

另一种策略是通过标准化降低复杂度。标准化几个硬件配置使机器可以互换。通常，为繁重的计算创建一个具备许多 RAM 和 CPU 的型号，以及一个具有许多磁盘的大存储量型号。现在，一个类别中的所有机器作为机器池处理，在需要时分配，不需要时归还机器池。可以创建一个 API，使这种分配就像创建虚拟机一样简单。

10.4.2 准备虚拟机

准备虚拟机应该由一次 API 调用实现。也就是说，建立集中标准规格可以使管理更加容易。

例如，可以按照规格分配 VM，在一台机器上容纳 4 个“小”VM、2 个“中”VM 或者 1 个“大”VM。之后，物理机器就不太可能出现有某些空间闲置、但不足以创建新 VM 的情况。

也可以使用斐波那契数的倍数。例如，如果一台 5 单元机器解除分配，就可以留出空间分配 5 台 1 单元机器或者一台 2 单元机器加上一台 3 单元机器，等等。

上述做法不仅有助于完全利用物理机器，还使它们的重组变得更容易。想象这样一种情况：需要分配一个“中”VM，但是在一台物理机器上只有容纳“小”VM 的闲置空间，另一台物理机器上还有一个“小”VM 的空间。如果 VM 的规格都是标准化的，就很容易确定如何移动 VM，在一台物理机器上创建“中”VM 规格的空间。如果每台 VM 的规格都

是定制的，仍然可以移动 VM，但是这时就像“河内塔”问题，需要许多中间步骤，效率低下。

10.4.3 安装 OS 和服务

安装操作系统和服务配置可以许多方式完成。两种主要的方法是映像方法和配置管理方法。

映像 (Image) 方法包括为每类服务创建一个磁盘映像。这个映像被复制到磁盘上，在重启之后，机器被预配置。映像可以部署到物理机器、虚拟机或者容器（在 3.2 节中描述）。映像包含操作系统、所有必要的软件包和做好运行准备的服务。这被称作**烤制映像 (Baked image)**，因为服务已经“烤制”在映像之中。

配置管理 (configuration Management) 策略包括使用安装程序或者其他机制启动运行一个最小化的操作系统。然后，配置管理工具可以安装任何附加软件包，启动服务。这种技术被称作“炒制”(Frying) 映像，因为映像在你等待的时候制作。

自动烤制

烤制映像的安装更快，因为所有配置预先进行。在调整数百台机器时，这显然是个优势。遗憾的是，维护许多映像可能成为负担。例如，在许多映像上安装某个补丁是劳力密集型的工作。这样完成的安装没有版本控制，这是很不好的。

解决方案是自动化烤制映像的创建。创建烤制映像的软件框架包括 Vagrant、Docker 和 Netflix Aminator。这些选择都提供描述如何从头开始构建映像的语言：指定基本 OS 发行版本、软件包、文件设置，等等。然后，根据这个描述创建映像。描述可以置于版本控制下，可使用服务交付平台构建、测试和部署映像。

烤制映像可用于构建新服务，也可以升级现有服务。例如，如果一个负载平衡器之后有 10 个 Web 服务器，升级工作包括从负载平衡器轮次中取出每个服务器，删除并从映像中重建。运行新发行版本的全部 10 个 Web 服务器都有易于理解的配置。

持久性数据

并不是所有机器都能按照上述方法清除重装。例如，数据库服务器或者文件服务器都有不可替代的数据，这些数据不是构建或者配置过程的一部分。这种情况下的解决方案是这些数据放到虚拟磁盘上，从存储区域网络 (Storage Area Network, SAN) 或者其他远程存储系统挂载。引导磁盘被替换，但是在启动时挂载虚拟磁盘，从而重新将系统与所需数据连接。通过将提供数据的位置与使用数据的位置解耦，机器的使用变得更加自由。

“烤制”与“炒制”的对比

配置管理往往比通过安装映像升级机器更快。配置管理的破坏性也较小，因为它只进行实现预期配置的最小限度更改。同样需要升级的 10 个 Web 服务器可以单独脱离负载平衡器轮次，仅对需要的软件包进行升级。利用这种技术，不需要清除机器，没有任何数据

丢失。

配置管理的缺点在于，机器的配置较难理解。想象加入第 11 个 Web 服务器的情况。它以新的软件发行版本启动。第 11 台机器的配置和经过升级的 10 台机器是否有相同的配置？从理论上是这样的，但是可能潜伏着小的差异。未来的升级需要覆盖两种情况的测试，增加了复杂度。烤制映像的支持者会提出，从一开始就刷新机器好过让混乱状态积累。

不同环境使用不同文件

在测试和生产环境中，往往有一组文件是肯定不同的。例如，图标和其他图像可能有专用于实际服务的特殊版本。还可能有特殊的凭据、证书或者其他数据。

处理这种情况的方法之一是在软件包中包含两组文件，使用配置管理将服务器指向正确的一组文件。但是，这种方法不适用于证书和其他可能需要暴露在所有环境之中的文件。

另一种解决方案是将环境特定文件移到单独的包中。每个环境有自己的配置包 (Configuration Package)。安装过程将安装主应用程序包和适合于该环境的配置包。

使用环境特定包的问题是这些文件本质上绕过了测试过程。因此，最好最大限度地控制这种机制的使用，最好将其限制在低风险的文件或者可以其他方式测试（如通过提交前检查）的文件。

10.5 持续交付

持续交付 (Continuous Delivery, CD) 是测试完全自动化且为每次构建触发运行的技术。对于每次构建，创建测试环境，运行自动测试，“交付”发行版本，做好在其他环境中使用的准备。这并不意味着每个更改都部署到生产环境，但是每个更改都证明可以在任何时候部署。CD 和持续集成有类似的好处。实际上，CD 可以视为对 CI 的扩展。CD 使小批量工作更加经济、风险更低，问题可以更快发现，从而更容易修复（参见 8.2.4 节）。

CD 包含持续集成的所有内容，加上系统测试、性能测试、用户验收测试和所有其他自动化测试。一旦测试完成自动化，实际上就没有任何借口不采用 CD。如果某些测试尚未自动化，CD 可以将发行版本交付到用于人工测试的 Beta 环境。

10.6 基础设施即代码

回顾图 9.1，服务交付平台 (SDP) 模式流程有代表基础设施和应用程序的象限。所有环境的基础设施应该通过自动化构建，这种自动化的处理就像 SDP 交付的任何其他服务一样。

这种方法被称作基础设施即代码 (infrastructure as code)。和应用程序代码一样，描述

基础设施的代码被保存在源存储库，接受版本控制，在测试环境中测试，然后经过批准部署于实际生产环境。

配置代码包括进行配置管理的自动化措施和任何配置文件及数据。配置管理代码和数据经过构建、打包等，正如应用软件一样。即使 VM 和容器的映像也可以构建和打包。

当基础设施即代码技术正确应用时，可以用相同代码构建开发、测试和生产环境。每个环境的不同之处仅在于使用的机器、副本的数量和其他设置。这最小化了测试和生产环境的差异，使测试更加精确。

任何人都应该能够构建开发或者测试的环境。这使团队和个人在运营和试验时无需干扰运营人员。开发人员和 QA 人员可以构建多个测试环境。SWE 可以用自己的便携电脑上的虚拟机构建自己的沙箱环境。

基础设施即代码随着硬件越来越虚拟化而变得更加容易。随着时间的推移，存储、机器和网络都会虚拟化。存储卷、虚拟机和网络拓扑可以通过软件创建、操纵和解除分配，并由 API 控制。

10.7 其他平台服务

SDP 中还涉及几种其他服务，值得在本章中做简单的介绍。

- ❑ **身份验证**：必须有某种形式的身份验证，使系统能够限制访问者。因此，需要提供身份验证、授权和计账（AAA）的安全设施。这种服务往往包括 LDAP 加上 Kerberos、Open Directory 或 Active Directory。
- ❑ **DNS**：DNS 将机器名称翻译为 IP 地址。在 SDP 中，机器的上线和下线都很快。通过 API 或者动态 DNS（DynDNS/DDNS）更新 DNS 分区的能力很重要。
- ❑ **配置管理数据库（Configuration Management Database, CMDB）**：部署阶段应该是数据库驱动的。配置和机器关系存储在数据库中，构建环境的工具使用这些信息指导它们的工作。这意味着修改环境就是简单地更新数据库。例如，数据库可能保存与特定服务相关联的 Web 服务器前端列表。在该列表中添加，配置管理工具就可以在那些机器上启动 Web 服务器。

10.8 小结

软件交付平台的部署阶段是软件包转化为运行服务的阶段。服务首先在测试环境中运行，在通过批准过程之后，使用软件包构建实际生产环境。

部署阶段涉及一组代表全部服务路径的软件包的选择。它们被视为一个发行候选，安装在组成环境的机器上。然后，服务经过配置，准备测试和使用。

环境需要基础设施：硬件、网络和其他组件。虚拟基础设施可以自动化配置。物理机

器也可以通过软件操纵，但是需要更多的规划、更慢也更不灵活。当基础设施配置在软件中编码时，整个基础设施可以作为软件对待，具有源代码的所有好处：版本控制、测试等。

在实际环境中使用某个发行版本之前，必须通过许多测试。系统测试检查整个系统。性能和负载测试验证软件的性能。用户验收测试是利益相关方批准发行版本的一个机会。发行可能还需要其他的一些批准，例如来自法律、市场和产品管理部门的批准。

持续交付是部署阶段和所有测试自动化之后实现的。新的发行候选自动制作并且随着测试的范围越来越大而增强信心。

服务交付工程是一个规模巨大、不断变化的领域。我们在本章中只是了解了一些皮毛，建议阅读《Continuous Delivery》(Humble & Farley 2010) 一书，更深入地了解这个主题。

练习

1. 描述服务交付平台部署阶段的步骤。
2. 描述持续交付及其优点。
3. 服务交付中有哪些常用测试？
4. 在你的组织的交付平台中，进行 10.2 节中的哪些测试，没有进行哪些测试？你希望通过增加遗漏的测试获得哪些好处？
5. 你的组织的服务交付过程中采用什么样的批准链？批准申请和响应如何传达？
6. 在你的组织中，如何自动化批准链？
7. 在没有进行软件部署的组织中应用本章中的部署阶段技术，但是选择使用现成商业化软件代替。
8. 在你的环境中，准备新机器有哪些步骤？哪些已经自动化？非自动化步骤如何自动化？
9. 你的组织是否采用持续交付？需要改变当前平台的哪些部分才能实现 CD？

升级运行中的服务

令你强大、为你带来某种成就感的一切都绝非易事。

——Jerri Nielsen 博士

本章介绍将新发行版本部署到生产环境的相关知识。这不同于任何其他环境中的部署。

用新软件发行版本升级某个环境的过程称作**代码推送**（code push）。将代码推送到生产环境可能很困难，因为我们将要修改正在运行的系统。这就像为在公路上降速到 90 公里/小时的汽车更换轮胎一样：你可以这么做，但是需要多加小心和细致规划。

幸运的是，有许多技术可以利用，每种适合于不同情况。本章分类介绍最常用的技术，然后，我们将讨论持续部署等最佳实践和其他实际问题。

11.1 卸下服务进行升级

升级服务的方法之一是将其卸下，将新代码推送到所有系统，然后重新启动服务。这样做的好处是很容易实施，而且可以在真实用户访问新升级的服务之前测试。

遗憾的是，这种技术需要停机时间，对于大部分服务来说这是无法接受的。但是，对于开发和演示环境是合适的，这些环境允许有事先计划的停机时间。

当服务完全复制的情况下，这种技术也是可行的。如果有足够的闲置容量，每个服务副本可以卸下、升级和恢复服务状态。在这种情况下，通常由一个全局负载均衡器（GLB）在工作副本之中划分流量。每次从 GLB 中移除（排空）一个副本，等待所有在途请求完成，然后，这个副本可以卸下，不会影响服务。

升级博客搜索

当 Tom 担任 Google 博客搜索 (Blog Search) 服务的 SRE 时, 面向客户的整个服务栈被复制到 4 个数据中心。每个副本独立于其他副本。整个系统有足够的容量, 任何一个栈可以下线, 由其他栈处理整个流量负载。每次可以从 GLB 中移除 (排空) 一个栈, 升级、检查之后加回 GLB 中。

同时, 系统的另一部分是“管道”: 扫描新博客帖子、吸收它们、生成新语料库, 并将其分发给 4 个面向客户的服务栈。管道对于整个服务非常重要, 但是如果它停止服务, 客户不会注意到。然而, 搜索结果的新鲜度将因为长时间的管道停机而恶化。因此, 正常运行时间很重要, 但不是必不可少的, 升级时将停止整条管道。

Google 的许多服务都以类似的方式架构, 升级的方式也类似。

11.2 滚动升级

在滚动升级中, 单独的机器或者服务器从服务中移除、升级并恢复服务。对每个待升级要素重复上述过程, 过程不断滚动直到全部完成。

客户看到的是持续的服务, 因为单独的停机被局部负载平衡器所掩盖了。在升级期间, 有些客户将会看到新软件, 有些客户则看到旧软件。由于连续的请求分别进入新旧机器, 有可能某位客户会看到新功能出现和消失。因为 4.2.3 节中讨论的负载平衡器黏性以及其它一些因素 (如 2.1.9 节讨论的部署新功能切换开关), 这种情况很少见。

在升级期间, 容量会有暂时性的减小。如果有 10 个服务器, 在每个服务器升级时服务的容量剩下 90%。因此, 这种技术需要经过规划, 确保有足够的容量。

这种过程的工作方式如下。首先, 排空服务器或者机器。这可以通过重新配置负载平衡器, 停止向其发送请求, 或者通过使副本进入“跛脚鸭模式” (2.1.3 节中描述过), 在这种模式中, 机器“说谎”——告诉负载平衡器自己处于不健康的状态, 使负载平衡器停止向其发送请求。最终, 在一段时间里不会接受任何新流量, 所有在途请求将会完成。接下来, 服务器升级、验证升级, 并撤销排空过程。然后, 下一台服务器的升级过程开始。

避免在困倦的时候进行代码推送

进行代码推送的最佳时间是白天。这时你很清醒, 如果出现问题也有更多的同事在场。

许多组织在深夜进行代码推送。在凌晨 3 点进行升级的典型借口是升级风险很大, 在夜间进行会降低曝光度。

在半睡半醒的状态下进行关键升级的风险要大得多。理想状况下, 我们现在已经使你深信, 降低风险的更好策略是自动化测试和小批量。

另外，你的团队所在地可能和进行代码推送的主位置有 8 个小时的时差。进行这种部署的时间对于客户来说是半夜，但是对你的团队却并非如此。

11.3 “金丝雀”

金丝雀 (Canary) 过程是滚动升级的一种特殊形式，更适合于需要升级大量元素的情况。如果有成百上千台服务器或者机器，滚动升级过程可能花费很长时间。如果每台服务器花费 10 分钟，升级 1000 台服务器将花费将近一周。这是无法接受的——但是一次升级所有服务器又太过危险。

金丝雀过程包括：升级极少数副本，观察是否有明显的问题发生，然后逐步升级更多的机器。在早期的煤矿开采中，矿工们会携带笼中的金丝雀进入矿井，这些鸟类对有害气体的敏感度远高于人类。如果金丝雀开始表现出病态，并从栖息的木杆上跌落，就应该离开矿井，避免因瓦斯而失去行动能力。

类似地，金丝雀技术升级一台机器，然后测试一段时间，问题往往发生在前 5 或者 10 分钟。如果金丝雀活着，则升级一组机器，再等待并进行更多测试，然后升级更多机器。

常用的金丝雀过程是：升级一台服务器，然后每分钟升级一台服务器直到所有服务器中的 1% 升级完毕，然后每秒升级一台机器直到所有机器升级完毕。在每两组之间可以有一个延长的暂停时间。此时，对所有已升级的服务器运行验证测试。这些测试通常很简单，只是验证代码不会崩溃，实际查询可以继续。

如果发现问题（也就是金丝雀死去），停止过程。此时可以回滚已升级的服务器。如果有足够的容量，也可以关闭这些机器直到新发行版本可用。

金丝雀过程不是测试过程，而是一种在生产环境中部署发行版本的方法，检测出有问题的代码推送，避免它们被用户发现。测试过程和金丝雀过程之间的差别是，测试过程失败是可以接受的。如果测试过程失败，你已经避免质量不佳的发行版本接触实际用户。更严格地说，测试过程已经在检测有缺陷发行版本上取得了成功，这是件好事。

相反，你不会希望金丝雀过程失败。失败的金丝雀过程意味着测试过程中遗漏了某些缺陷。失败的金丝雀过程应该非常少，它会造成停止开发，投入资源确定问题所在以及在未来避免这种问题所需要添加的附加测试。只有这样，才能开始新的试运行工作，金丝雀是对因为疏忽而发行质量不佳版本的保险策略，而不是检测发行质量不佳的手段。

测试和金丝雀过程往往被合为一体，但是不应该这么做。有些人所称的金丝雀过程实际上是在实际用户身上测试新发行版本。测试应该在测试环境中进行，而不是在实际用户身上进行。

金丝雀过程不能代替系统测试

我们已经观察到在一些情况下，金丝雀过程被用于在实际用户身上测试新发行版本。在一种情况下，这是无意的——在发生重大运行中断之前没有意识到的一个事实。SRE 接受全面测试过的软件包，并采用金丝雀过程部署到生产环境。这个软件在许多年间都工作得很好，因为测试和实际环境非常类似。但是，随着时间的推移，SRE 开发了许多用于生产环境的工具。这些工具没有由开发人员的测试系统进行测试。开发人员不负责这些工具，而且许多工具被认为是专用或者临时性的。

工程学上有一句老话：“没有什么比临时性的解决方案更有持久性。”这些工具很快成长并形成复杂的自动化系统。然而，它们没有经过开发人员测试系统的测试。每个重大的发行版本都会破坏这些工具，运营人员不得不匆忙更新它们。但是，和下面要发生的问题相比，这不算什么。

有一天，一个发行版本投产，在推送完成之前未发现问题。特定用户的服务中断了。

现在，两个环境中使用的硬件有了很大的差异，因此内核驱动程序和虚拟化技术版本都有差异。结果是运行某些操作系统的虚拟机停止工作。

到这个时候，SRE 们意识到两个环境的差异已经过大。他们必须全面改造系统测试环境，确保测试生产环境中使用主服务发行版本、内核版本、虚拟化框架版本的特定组合。此外，他们必须将自己的工具加入存储库和开发及测试过程，这样就不用每次都慌乱地用工具修复自己所开发环境的不兼容性。

创建合适的系统测试环境以及保持测试 / 生产环境同步的机制，需要许多个月的努力。

11.4 分阶段试运行

另一种策略是将用户分为几组，每次升级一组。每组（阶段）由对风险的宽容度标识。

例如，Facebook 有专门为自己的雇员提供服务的群集。这些群集首先接受升级，因为他们的雇员愿意作为新发行版本的测试者——这是工作的一部分。接下来，升级一小组外部用户群集，最后升级剩下的群集。

Stack Exchange 的升级过程包括许多阶段。Stack Exchange 有 110 多个 Web 社区，每个社区还有一个讨论社区本身的关联元社区。所有社区使用相同的软件，但是颜色和设计不同。部署阶段是测试环境，然后是元社区，再往后是人数较少的社区，最后是最大、最活跃的社区。每个阶段在前一阶段运行一定时期内未发现任何问题时自动开始。到升级的最后阶段，Stack Exchange 对发行版本已经有很高的信任度。最早的阶段可以忍受更多的运行中断，这有许多原因，其中包括它们不是产生收入的单位这一事实。

11.5 按比例分片

按比例分片 (proportional shedding) 是一种部署技术, 新服务在与旧服务并行的新机器上构建。然后, 负载均衡器发送 (或者分片) 一小部分流量到新服务。如果成功, 则发送更高比例的流量。这一过程持续到所有流量发送到新服务为止。

按比例分片可用于在两个系统之间移动流量。旧群集在整个过程完成之前不关闭, 如果发现问题, 负载可以切换回旧群集。

这种技术的问题在于过渡期间需要两倍容量。如果一台机器可以容纳服务, 在升级期间有两台机器是合理的。

如果有 1000 台机器, 按比例分片可能非常昂贵。保留 1000 台备用机器可能超出预算。在这种情况下, 一旦某一比例的流量切换到新群集, 一些旧机器可以回收, 重新部署为新群集的一部分。

11.6 蓝 - 绿部署

蓝 - 绿部署类似于按比例分片, 但是不需要两倍的资源。在相同的机器上有两个环境, 一个称为“蓝”, 另一个称为“绿”。绿环境是实际环境, 蓝环境处于休眠。通过某种机制 (简单的机制是有两个不同子目录, 每个作为同一个 Web 服务器的不同虚拟主机), 两者存在于同一台机器上。蓝环境消耗的资源很少。

新发行版本上线时, 流量被导向蓝环境。当过程完成之后, 环境名称互换。这种系统可以很容易地回滚到前一环境。这是应用程序零停机时间部署的一种非常简单的途径, 这些应用程序不需要专门设计, 只需要在同一台机器上的两个不同位置安装应用程序支持。

11.7 功能切换

正如 2.1.9 节中所讨论的, 将每个新功能绑定到某个软件标志或者配置设置, 使这些功能可以单独开关, 是一种常见的做法。这种开关的切换称作**标志翻转 (Flag Flipping)**。不完整和尚未为实际用户使用做好准备的新功能标记为“关闭”。当它们准备就绪, “开启”标志。如果发现问题, 可以禁用这些功能。“关闭”标志也称作“将功能藏在标志之后”。

功能切换开关是实现部署和发行的解耦一般原则的一种方法。部署是将软件包投入某个环境, 发行是使某项功能可用于用户。我们往往通过部署实现发行, 因此认为两者的含义相同。实际上, 功能开关将两个概念解耦, 这样部署可以发生在任何时候, 而发行按需发生。部署变成用户并不知晓的纯技术活动。

实现标志机制有许多方法。标志可以是命令行标志, 在启动服务时使用。例如, 运行如下命令, 可以启动某个拼写检查器服务第 2 版, 启用拼写检查算法、禁用形态学算法:

```
$ spellcheck-server --sp-algorithm-v2 --morphological=off
```

标志可以通过 Shell 环境变量设置。这些变量在运行命令之前设置：

```
$ export SP_ALGORITHM_V2=yes
$ export SP_MORPHOLOGICAL=off
$ spellcheck-server
```

随着标志的增加，这种方法变得笨拙。在任何时候，服务可能有几十个甚至数百个标志。因此，标志系统可以从文件读取标志代之：

```
$ cat spell.txt
sp-algorithm-v2=on
morphological=off
$ spellcheck-server --flagfile=spell.txt
```

这些方法都会碰到这样一个问题：要更改任何标志，过程必须重启，从而中断服务。其他技术可以任意更新标志。Google 的 Chubby 和 Apache Zookeeper 项目等系统可以用于存储标志，在它们变化时高效地通知数千台服务器。利用这种方法，只需要一个标志——指定 zookeeper 命名空间的标志——就可以找到配置：

```
$ spellcheck-server --zkflags=/zookeeper/config/spell
```

使用标志翻转有许多理由：

- ❑ **快速开发：**为了实现快速开发，功能由对源代码主分支的一系列小更改构建而成。大的功能所需的开发时间较长，将代码更改合并到源代码主线的等待时间越长，合并的难度和风险就越大。有时候，源代码可能被其他开发人员更改，造成合并冲突。合并少量代码出错的可能性较小（更多细节参见 8.2.4 节）。考虑到这一事实，不完整的功能通过在准备就绪之前禁用标志来隐藏。在某些环境中，标志可能比其他环境更早启用。例如，在演示环境中启用标志，可以让产品管理团队预览某项功能。
- ❑ **逐步引入新功能：**一些功能往往先向某些用户推出，而不向其他用户介绍。例如，Beta 用户可能较早访问新功能。当用户登录时，通过调整标志动态控制对 Beta 功能的访问。在这种技术发明之前，Beta 用户转向完全不同的一组服务器，设置仅支持 Beta 用户的完整服务器环境副本的代价很高。而且，控制粒度是“要么全有，要么全无”。有了标志，每个单独功能可以进入或者移出 Beta 测试。
- ❑ **精细安排的发行日期：**排定标志翻转的时间比排定软件部署时间更容易。如果要在周二中午公布某项新功能，精确地在那个时刻开始试运行新软件很困难，尤其是在有许多服务器的情况下。相反，标志可以动态控制，在特定时间翻转。
- ❑ **动态回滚：**通过禁用标志禁用表现不佳的新功能，比回滚到软件的旧版本更容易。如果新发行版本有许多新功能，因为某个功能不好而回滚整个发行版本是羞耻的事情。有了标志翻转，可以只禁用一项功能。

- ❑ **缺陷隔离**：让每处更改与一个标志关联，有助于隔离缺陷。想象一下 30 项最近新增功能之一可能造成内存泄漏的情况。如果它们都与切换开关相连，二进制搜索可以识别出造成问题的功能。如果二进制搜索无法在测试环境中隔离问题，通过标志在生产环境中进行缺陷检测比通过许多单独发行版本要合理得多。
- ❑ **A-B 测试**：了解用户更喜欢某种设计还是其他设计的最佳手段往往是两者都实现，向某些用户展示新功能，观察它们的表现。例如，假定某项产品的签约数量很低。如果某个复选框默认选中或者使用完全不同的机制代替复选框，签约数量会不会提高？这时可以选择一组用户，一半采用复选框默认选中（A 组），另一半使用新机制（B 组）。在测试之后，所有用户都将使用获得更好结果的设计。在测试结束时，可以使用标志翻转控制测试，启用胜出的设计。
- ❑ **1% 测试**：这种测试向用户的一个统计样本揭示某项功能。有时候，和金丝雀过程类似，用于在全局部署新功能之前的测试。有时候，和 A-B 测试类似，用于识别对试验性功能的反应，然后决定是否应该部署。有时候，这种测试用于通过统计采样收集信息。例如，网站可能想要收集有关页面加载时间的性能统计。可以在 HTML 页面中插入 JavaScript 代码，传回关于页面加载时间的服务数据。收集这些数据本质上使服务查询增加了一倍，将使系统超载。因此，可以仅在 1% 的时间内插入 JavaScript。标志设为 0 可以禁用收集，或者设置为指定采样比例的某个值。
- ❑ **差异化服务**：有时候，有必要对不同用户启用不同服务。好的标志系统可以让付费客户看到不同于未付费用户的功能。与一组标志关联，能够实现许多会员级别。

有时候，使用标志禁用与某项新功能有关的一切。另一些时候，只是用于隐蔽其可见性。假定某网站打算为一个输入字段添加自动完成功能，这一功能需要更改收集输入提示的 HTML、对部分查询添加一个新的数据库 API 调用，可能还需要添加其他代码。标志可以禁用与该功能相关的所有代码，或者简单地控制该功能在 UI 中是否出现。

案例研究：Facebook Chat 的摸黑启动

2011 年，Facebook 的 Chuck Rossi 做了题为“Pushing Millions of Lines of Code Five Days a Week”（在一周的 5 天中推送数百万行代码，Rossi, 2011）的演讲，描述了管理摸黑启动（Dark Launch，在生产环境中投放用户不可见组件的代码）的内部工具“Gatekeeper”。Rossi 的演讲揭示了这样一个事实：在 Facebook.com，任何时刻都有用于在下 6 个月甚至更久的时候将要启动的重要功能的代码。Gatekeeper 可以对向哪些用户揭示哪些功能提供细粒度的控制。有些过滤器很明显，如国家、年龄和数据中心。其他过滤器是不可预知的，包括排除某些媒体出口（如 TechCrunch）的已知雇员（Siegler, 2011）。（参见 18.4 节的相关案例研究）

11.8 在线模式更改

有时候，新软件发行版本需要采用与前一发行版本不同的数据库模式。如果服务可以承受停机时间，就可以停止服务，升级软件并更改数据库模式，然后重启服务。但是，在 Web 环境中停机时间几乎是不能接受的。

在典型的 Web 环境中有许多 Web 服务器前端（或者副本），它们都与同一个数据库服务器通信。旧的软件发行版本无法理解新数据库模式，将会发生故障或者崩溃。新的软件发行版本无法理解旧数据库模式，也会失效。因此，不能在更改数据库模式之后升级软件：数据库修改后，旧的副本立刻会失效。在副本升级之前，服务无法使用。也不能先升级副本再更改数据库模式，因为所有升级后的副本将会失效。在滚动升级期间，模式无法升级：新副本将会失效，在数据库模式更改时，所有失效的副本开始工作，而正常工作的副本会开始失效。多么混乱啊！而且，这些方法在发生问题时都没有合适的回滚手段。

处理这种情况的方法之一是使用数据库视图（Database View）。每个视图提供对同一数据库的不同抽象。每个新软件版本编码一个新的视图，将软件升级与模式更改解耦。现在，当模式更改时，每个视图的代码必须更改，提供新模式的相同抽象。模式中的更改和视图代码的升级是原子操作，实现了平滑的升级。例如，如果一个字段以新格式存储，一个视图将以新格式存储，另一个视图将在旧格式和新格式之间转换。当模式更改时，视图的角色将颠倒。

遗憾的是，这种技术不太经常使用。视图在关系数据库中是一项稀有功能。在本书编著期间，没有任何键-值存储（NoSQL）系统支持视图。即使这种功能可用，也不总是使用。在这些情况下需要另一种策略。

替代方案之一是在两个软件发行版本期间更改数据库模式：一次是在数据库中添加新字段之后，另一次是在删除过期字段之前。我们从 Stephen McHenry 那里学到这种技术，将其称作 McHenry 技术。类似的技术被称作“扩展/收缩”。

下面是这种技术的各个阶段：

- 1) 运行代码读取和写入旧模式，只从表或者视图中选择需要的字段。这是原始状态。
- 2) 扩展：添加新字段修改模式，但是没有删除任何旧模式。对代码不做任何更改。如果需要回滚，因为还没有使用新字段，不会造成麻烦。
- 3) 修改代码使用新模式字段并投产。如果需要回滚，只需要恢复到第 2 阶段。此时，可以在系统运行中进行任何数据转换。
- 4) 收缩：删除引用现已不用的旧字段的代码并投产。如果需要回滚，只须恢复到第 3 阶段。
- 5) 从模式中删除现已不用的旧字段。此时回滚不太可能发生，如果需要，数据库将简单地恢复到第 4 阶段。

这种技术的一个简单例子涉及保存用户档案文件的数据库。假定要添加存储用户相片的能力，在模式中添加存储照片信息的新字段（第 2 阶段）。投放一个软件发行版本以处理

新字段，包括字段为空的情况。下一个软件发行版本删除遗留代码和标志需求。因为模式更改只是添加字段，不需要第 5 阶段。

再举一个例子，假定用新字段替换某个字段。用户的全名保存在一个字段中，新模式使用 3 个字段分别保存名、中间名和姓。在第 2 阶段，这 3 个新字段添加到模式。在第 3 阶段，我们推出一个软件发行版本，读取旧字段和新字段。代码根据填充的旧字段或者新字段完成对应的操作。用户的更新写入新的单独字段并将旧字段标记为不使用。此时，可以运行一个批量作业，找出仍然使用旧字段的所有档案，并将其转换为新字段。

一旦所有遗留数据转换完毕，删除任何使用旧字段的代码。新发行版本投产（第 4 阶段），部署这个版本之后，第 5 阶段从数据库模式中删除旧字段。

简化上述过程的一种方法是合并或者重叠第 4 阶段和第 5 阶段。另一种优化措施是延迟旧字段的删除——可能在下一次模式更改时进行。换言之，版本 n 的第 5 阶段与版本 $n+1$ 的第 2 阶段合并。

在第 2 阶段中，软件将会发现新字段，这可能造成问题。必须编写软件忽略新字段，在存在意外字段时不会中断。一般来说这不成问题，因为大部分 SQL 查询请求所需的精确字段，访问 NoSQL 数据库的软件倾向于按照惯例进行这种操作。按照 SQL 的说法，这意味着 `SELECT *` 的任何使用不应该假定所接收数据字段的数量或者位置。这通常是一种好的做法，因为它使你的代码更加健壮。

11.9 在线代码更改

有些系统允许在线代码更改，这使性能升级更加容易。我们通常对修改在用系统的技术并不满意，但是有些语言专门设计以支持这种能力。

Erlang 就是这类语言。以 Erlang 编写的服务可以在运行中升级。构造合理的 Erlang 程序设计为事件驱动有限状态机（FSM）。对于接收的每个事件，调用特定函数。服务可能接收的事件之一是代码已经升级的通知。调用的函数负责升级或者转换任何数据结构。所有后续事件将触发函数的新版本。Erlang 引擎本身重启时没有任何过程能够幸存，但是精心计划的升级可以实现 0 停机时间。

在 <http://learnyousomeerlang.com/what-is-otp> 上可以找到对 Erlang 基本功能和结构非常易于理解的介绍。

我们期待着在未来看到更多这样的系统。在《星际迷航：下一代》中，Geordi 编辑在用代码。我们乐观地认为，有一天我们也能做到这一点。

11.10 持续部署

持续部署意味着通过测试的每个发行版本都自动部署到生产环境。持续部署应该是大

部分公司的目标，除非受限于外部监管或者其他因素。如图 11.1 中的描述，这需要持续集成和持续交付，以及其他测试、批准和代码推送过程的自动化。

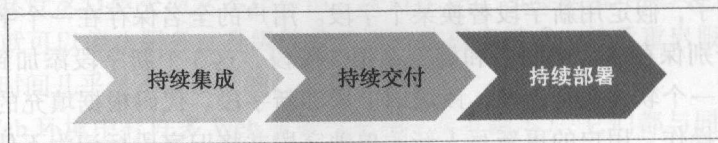


图 11.1 持续集成、交付和部署互为基础

回忆一下，持续交付产生投产就绪的软件包，但是否真正地将每个软件包部署到生产环境是一个业务决策。持续部署是自动化这种批准过程并始终部署已批准发行版本的业务决策。

不断推送代码听起来很危险，实际上也是如此。因此需要采取许多防范措施，其中之一是仅在非关键环境（如 Beta 和 UAT 环境）中使用持续部署。

另一种防范措施是在生产环境中采用这种方法，但是仅在对其发行已有很高信任度的特定子系统上使用。例如，在提供 API 和后端服务的子系统中常常看到持续部署，但是 Web UI 子系统仍然人工测试。这是面向服务架构的另一个好处。

即便如此，有些人可能仍对持续部署感到不安。作为有经验的系统管理员，你可能有多种“直觉”，帮助你决定今天是不是代码推送的好时机。例如，你可能采用在关键日期中不进行推送的策略，例如在财务部门关账时，运行中断的后果特别严重。你可能注意到近来不完整的构建过程数量大得异乎寻常，因而意识到开发人员这一周不太好过。这使你对于接受发行版本保持警惕，必须首先和首席开发人员交谈，了解他们是否有很大的压力。考虑这些问题是合理的做法。

对于某些人，这些直觉可能成为避开持续部署的借口。实际上，直觉能够提供接受部署的路线图。所有这类直觉都有一定的真实性，那么为什么不将它们转换为自动禁用自动化部署的可计量指标？

下面是决定是否暂停持续交付时应该考虑的一些因素。其中一些因素似乎是只有人才能够确定的。但是，利用一些创造性，它们就都可以自动化，作为自动化代码推送的门限：

- ❑ **构建健康**：最近的构建失败往往代表着其他稳定性问题。如果最近的 m 次构建中有 n 次失败，可以停止自动化推送。这可以作为计量开发人员压力或者匆忙工作的代用指标。
- ❑ **测试全面性**：只有经过充分测试的代码才应该自动推送。“代码覆盖率”等指标可以作为测试是否宽松的指标。
- ❑ **测试可再现性**：有时候通过、有时候失败的测试被称作“古怪”的测试。这种情况可以通过多次、随机顺序运行测试检测。如果测试可再现性很低，可以自动暂停交付。
- ❑ **生产健康**：任何单一测试都无法表明系统健康，但是监控系统没有明显的警报是很

好的代用指标。

- ❑ **时间表许可**：如果有事先计划的、因为假日、季度财务报告或者过多关键人物休假造成的“更改冻结”，禁止进行自动推送。可以简单地用暂停部署日期列表覆盖这些情况。
- ❑ **值班安排**：为了避免将当班的工程师从床上叫起来，在典型的睡觉时间可以暂停推送。如果有多个值班团队，每个负责一天中的不同时间，他们应该有不同的清醒时间。即使三班倒的全天候排班，也会有无人完全清醒的时段。
- ❑ **人工停止**：应该准备可以单击某个按钮停止自动化推送的人的名单。这类似于装配线，如果发现次品，任何人都可以停止生产。人工暂停的原因不一定是紧急状况。
- ❑ **推送冲突**：服务可能由许多子服务组成，每个子服务都有自己的发行计划。一次只允许部署一个子服务是明智的做法。同样，如果当前部署没有结束，不应该开始下一次推送。
- ❑ **有意推迟**：在推送之间暂停，让当前推送“吸收”可能很有益处——运行足够长的时间，验证其稳定性。过快推送可能会使问题开始时的隔离变得困难。
- ❑ **资源争用**：如果资源不足——例如，磁盘空间不足或者 CPU 利用率奇高——应该暂停推送。负载必须低于特定的阈值：在系统被“淹没”时不要推送。必须有足够的冗余度：如果副本不是 $N+2$ 配置，不要推送。

将这些直觉变成自动化过程似乎很危险。真相是，将其自动化并始终执行，比让人们去凭直觉决定停止推送更安全。运营应该根据数据和科学进行，自动化这些检查意味着不管谁正在休假，它们每次都一致地运行。可以对这些过程进行微调和改进。我们曾经多次听到人们说，运行中断的发生是因为他们出于懒散而决定不进行某种检查。事实是自动化检查可以改善安全性。

在实践中，人们对退化的捕捉能力绝不比自动化测试强。实际上，相反的想法是荒谬的。例如，人们通常注意到的退化之一是服务需要的 RAM 或者 CPU 明显比前一版本多。这常常是内存泄漏或者其他编码错误的标志。即使自动化系统检测到这些情况并提出警告，人们也往往忽略这些问题。持续部署如果实施得当，就不会忽略这些问题。

实施持续部署并不轻松，如果从新项目开始、规模还小时就开始会更容易。人们也可以采用对任何新子系统持续部署的策略。旧系统最终会退役，或者在它们停止发展时添加持续部署。

11.11 处理失败的代码推送

尽管我们进行了所有测试且提出了苛刻的要求，有时候代码投产仍会失败。有时候，代码发生硬故障——软件拒绝启动或者在启动之后立刻失效。理论上，金丝雀过程应该检测出硬故障，并将发生故障的副本移出生产环境。遗憾的是，并不是所有服务都进行了复

制，或者以可采用金丝雀过程的方式复制。在其他情况下，故障更加细微。例如，功能的灾难性失效可能不会被马上注意到，也无法通过标志或者其他技术缓解。结果是，我们必须更改软件本身。

方法之一是回滚（roll back）到最后一个已知的“好”发行版本。发现问题时，卸载软件，重新安装最新的“好”发行版本。

另一种方法是前滚（roll forward）到下一个发行版本，该版本可能已经修复了失败发行版本之中发现的问题。这种技术的问题是下一发行版本可能需要几个小时或者几天才能出现。失败的发行版本必须有足够的弹性以在此期间使用，或者必须有可用的变通手段。第6章的弹性技术可以减少涉及的时间压力。希望采用这种技术的团队必须专注于缩短 SDP 代码等待时间，直到这个时间足够短，使前滚成为可能。

前滚在服务器高度复制并使用金丝雀过程部署时最有效。灾难性故障（如服务器不能启动）应该在测试环境中找出。如果因为某种原因没有找出问题，第一批“金丝雀”应该出现故障，从而阻止其他副本升级。在正常工作的版本部署之前，容量将略有降低。

回滚的批评者们指出，真正的回滚是不可能实现的。卸载软件并重装已知的好发行版本仍然是更改。实际上，这种更改很可能没有在生产环境中测试过。在生产环境中进行未经测试的处理是应该不惜一切代价避免的。将其作为应急措施意味着，在最不希望出现风险的时候做高风险的事情。

前滚有压倒性的优势，持续部署环境为以较小风险实现前滚提供了信心。实话说，有时候前滚是不可能实现的。因此大部分网站使用混合解决方案：在可能的情况下进行前滚，在不得已的情况下回滚。而且，在这种情况下，有时候推送能够修复难以克服问题的小更改更为方便。这种紧急热修复（emergency hotfix）的风险很大，因为通常没有接受全面测试。紧急修复和回滚应该小心跟踪，催生出消除它们的项目。

11.12 发行原子性

发行涉及特定组件还是整个系统？

松散耦合的系统包含许多子系统，每个都有自己的版本序列。这增加了测试的复杂度，测试所有版本的组合很困难，往往不可能完成。这样做常常是对资源的浪费，因为并不是所有组合都会使用。

一种技术是测试组件版本的特定组合：例如，服务 A 运行发行版本 101，服务 B 运行发行版本 456，服务 D 运行发行版本 246。“（101+456+246）元组”被作为一组测试、批准和部署。如果测试失败，改进一个或者多个组件并从头开始测试新的元组。其他组合可能有效，但是我们开发一个已经批准的组合列表，仅使用那些组合。如果需要回滚，则回滚到前一个批准的元组。

当组件紧密耦合时，上述技术通常是必需的。例如，负载均衡器软件、插件和使用的

OS 内核在性能退化可能造成毁灭性影响的高要求环境中往往作为一个元组测试。另一个例子是虚拟化管理系统的高耦合组件，如 Ganeti 或者 OpenStack；虚拟化技术，如 KVM 或者 Xen；存储系统（DRBD）以及 OS 内核。

这种技术的问题在于降低了潜在的变化率。由于所有组件同步发展，其中一个延迟发行意味着已经做好前进准备的组件不得不等待。

如果组件是松散耦合的，每个组件可以独立测试并按照自己的速度推进，问题也隔离得更好。更改后的组件可能失败，在这种情况下我们知道问题出在该组件上。其他组件也可能失败，在那种情况下我们知道更改后的组件引入了不兼容的情况。

这样的系统只在组件松散耦合时有效。例如，在 Google，整个基础架构是松散耦合的小型服务组成的生态系统。服务不断升级，要求整个公司停下来测试系统是不可能的。Google 的基础架构更像一个生物系统而非机械钟。因此，每个服务必须严格实现向上兼容。不兼容的更改提前很久宣布。专门创建的工具帮助管理和跟踪这些更改。例如，当 API 将以不兼容的方式更改，有一种手段检测很快将被废弃的使用，并对受到影响的服务所有者提出警告。

11.13 小结

升级在某个环境中运行的软件称作代码推送。代码推送需要规划，以及专为升级运行中服务设计的技术。

有时候，“后台运行”的服务可以停止进行升级。在更多时候，服务的升级方式是停止各个部分，升级并恢复运行（即滚动升级）。如果有许多副本，可以将其中一个指定为“金丝雀”，在其他副本升级之前作为测试。按比例分片缓慢地将流量从旧系统迁移到新系统，直到旧系统不接受任何新流量。

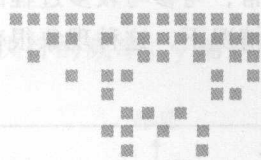
标志翻转技术在发行版本中包含新功能但是将其禁用，在指定的时候用一个标志或者软件切换开关启用这些功能。这在发现问题时容易恢复更改：简单地关闭切换开关。

在实际系统上更改数据库模式很难协调。不可能在完全相同的时间内更改所有客户端软件和数据库模式。作为替代，可以使用 McHenry 技术解耦两者的更改。添加新字段、升级软件以使用旧字段和新字段、升级软件只使用新字段，然后从模式中删除旧字段。

当推送失败时，有时候可以前滚到下一个发行版本，而不是恢复到最后一个已知的好发行版本。恢复代码的风险很大，失败的回滚是比失败的推送更大的问题。持续部署是自动化代码推送、在某些条件表明风险过高或者业务条件需要暂停时自动暂停或者推迟的过程。和持续集成及持续交付一样，通过频繁地以自动化形式完成任务，我们降低了风险、改进了过程。

练习

1. 总结升级在用服务软件的不同技术。
2. 为什么金丝雀过程的失败是悲剧性的，部署应该停止，直到找到根源？
3. 你的当前环境中使用本章的那种推送技术？
4. 你当前环境应该采用哪一种推送技术？它们有什么优点？
5. 在你的环境中如何使用暗启动？
6. 什么是持续部署，有何好处？
7. 为什么不鼓励回滚？如何更安全地进行回滚？
8. 如何在你的环境中开始实施持续部署？如果不这么做，原因是什么？
9. Jerri Nielsen 博士是谁？



自动化

逐步改进胜过久拖不决的完善。

——马克·吐温

自动化是让计算机为我们工作。历史上，对自动化的渴望受到 3 个主要目标的激励：更精确、更稳定、更快。其他因素如提高安全性、提高容量和降低成本则是这 3 个基本目标令人满意的副作用。作为系统管理员，自动化需要完成的任务，应该占据大部分工作量。我们不应该对系统做出更改，而是应该命令自动化系统完成这些更改。

手工劳动的回报是线性的，也就是说，执行一次就得到一次的好处。相比之下，花在自动化上的时间将从代码的每次使用中得利。多次使用代码，回报就会倍增。

对运行中的系统进行更改应该涉及对保存在版本控制中央存储库中的代码和数据文件的更改。然后，这些更改由软件交付平台选择、测试并部署到生产环境。要以这种方式工作，现代系统管理员必须是软件开发人员。自从安装新计算机成为一个 API 调用，我们都成了程序员。这种脚本不是艰深的计算机科学，也不需要正式的培训。随着时间的推移，它导致我们学习越来越多的软件开发技能（Rockwood 2013）。自动化不会让系统管理员失业。确实，总是有更多的工作要做，能编写代码的系统管理员对于雇主来说更有价值。实际上，因为系统管理只能通过自动化扩展，不知道如何编码会使你的职业生涯充满风险。

然而，自动化不应该通过简单地开发比人工任务的更快、更可预测的功能性替代品来实现。人-机系统必须作为一个整体来考虑。很少有能够完全自动化的系统。大部分系统会遇到无法自动化处理的异常现象，这些现象必须由人来处理。但是，如果运行自动化系统的人对其工作原理毫无了解，就难以知晓故障的情况和原因，因而非常难以

恰当地处理异常，与参与较多过程的人有着很大的差距。在处理异常的人并非编写自动化系统的人的情况下，这种现象很普遍。在某种程度上，这是所有自动化系统都存在的现象。

必知术语

领域特定语言 (Domain-Specific Language, DSL)：为特殊用途（如系统管理、数学或者文本操纵）专门构建的语言。

苦活：令人疲劳的体力劳动。

12.1 自动化方法

自动化设计有 3 种主要方法。第一种（最常见的）方法是用“剩余”（left-over）原则自动化——自动化处理尽可能多的任务，人们处理所有余下的工作。第二种方法是“补偿”原则，根据所擅长的任务在人和自动化系统之间分割工作。第三种方法基于互补原则，旨在改善人机结合系统的长期健康。

自动化的影响超出了直接的预期目标。理解自动化可能带来的预期外效果，可能帮助你构建更健康的整体系统。例如，防抱死刹车（Antilock Brake, ABS）使司机更容易在紧急情况下快速停车。单独来说，这种自动化措施应该减少事故。但是，这种自动化措施的出现影响了人们的驾驶方式。他们在湿滑的条件下比没有 ABS 时开得更快，而且与前车的距离更短，因为他们对 ABS 自动化有信心。这些就是自动化的预期外结果。当人为因素没有考虑在内时，自动化可能产生意外（可能是负面）的结果。

记住，并不是所有任务都应该自动化。有些艰难的任务很少见，只发生一次便不再发生，例如，安装一个复杂的大型软件系统。这是应该外包的任务，可以雇佣以前实施过这项任务的顾问，因为这种情况很少，学习曲线完全没有回报。顾问的工作成果应该包含一组更频繁执行的常见任务，可以移交给常规运营人员或者自动化系统。

还要记住一点，不管任务是容易、困难、频繁还是罕见，消除它们都有好处，因为需要了解、完成或者维护的事情又少了一件。如果可以消除任务而不是实施自动化，那将是最为高效的方法。UNIX 和 C 语言的创始人之一 Ken Thompson 曾经写过著名的一段话：“在我最有效率的一天中，抛弃了 1000 行代码。”

12.1.1 剩余原则

运用剩余原则，我们自动化所有可以合理自动化的任务，剩下的人工处理：过于罕见或者过于复杂而无法自动化的情况。这种观点造成了不现实的假设：人具有无限的通用性和适应性，能力没有极限。剩余原则首先考察人们所完成任务，用一系列因素决定需要自动化的任务。在系统管理领域中，运营人员可以做许多事情。这些任务可以沿着图 12.1

中的坐标轴分类。

x 轴的标签从“罕见”到“频繁”，表示任务执行的频率。y 轴的标签从“容易”到“困难”，表示每次执行任务需要的工作量。

- ❑ 分类为罕见 / 容易的任务可以继续人工进行。如果很容易，任何人都可以成功地执行。团队的文化将影响人们能否做正确的事情。
- ❑ 分类为罕见 / 困难的任务应该记入档案，创建工具辅助该过程。文档和更好的工具使正确和一致地完成任务更加容易。这一象限包括无法自动化的故障检修和恢复任务。但是，好的文档可以协助该过程，好的工具可以消除重复劳动的负担或者人为错误。

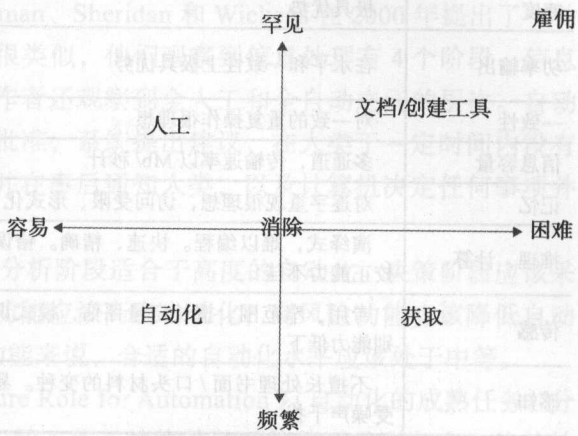


图 12.1 任务可以根据工作量和频率分类，这决定了下一步如何优化它们

- ❑ 分类为频繁 / 容易的任务应该自动化。投资回报显而易见，有趣的是，一旦某项任务有了文档，就会变得更容易完成，从而滑向这一象限。
- ❑ 分类为频繁 / 困难的工作应该自动化，但是最好是获取自动化措施而非自行编写。购买商业化软件或者使用免费 / 开源项目，可以利用千百人的技能和知识。

剩余原则的目标是通过实施任何可行的自动化，达到高效率，没有明确地考虑人为因素。但是，这一原则很容易描述，决定需要自动化的任务也相对容易。即使不是唯一考虑的原则，也可以应用从这种方法得到的经验教训。特别是，在“容易 - 困难”和“罕见 - 频繁”坐标轴上考虑任务，是研究自动化的有用工具。

12.1.2 补偿原则

补偿原则基于 Fitts 列表，这种列表的名称来源于 Fitts (1951)，他提出了一组决定自动化内容所用的属性。这些属性如表 12.1 所示。尽管 Fitts 的工作已经过去了 60 多年，这些属性仍然适用。

这条原则基于人和机器的能力保持静态的假设，相应地分割工作。这种方法不是隐舍地将人视为无限通用的机器，而是旨在避免对人提出过分的要求。

采用补偿原则，我们将确信机器比人更适合于从数千台机器中以 5 分钟的间隔收集监控数据。因此，我们将自动化监控。我们还可以确定，人无法在高毒性的核事故现场存活，但是设计得当的机器人却能做到。

表 12.1 Fitts 列表原则

属 性	机 器	操作员 / 人
速度	极具优势	相对慢，以秒计
功率输出	在水平和一致性上极具优势	相对弱，峰值大约 1500 瓦，在工作日期间低于 150 瓦
一致性	对一致的重复操作很理想	不可靠，受制于学习（习惯）和疲劳
信息容量	多通道，传输速率以 Mb/ 秒计	主要是单通道，低传输率 <10b/ 秒
记忆	对逐字重现很理想，访问受限、形式化	适合于原则和策略，访问全能且有创造力
推理，计算	演绎式，难以编程。快速、精确。错误校正能力不佳	归纳式，易于编程。缓慢、不精确。出色的错误校正
传感	专用、窄范围。擅长定量评估，模式识别能力低下	宽能量范围，多功能。擅长模式识别
感知	不擅长处理书面 / 口头材料的变种。易受噪声干扰	擅长处理书面 / 口头材料。易受噪声干扰

根据这一原则，人被视为信息处理系统，工作用人机交互描述，每项工作都有不同的可辨识任务。补偿性原则关注人 - 机交互的优化，使过程更加高效。

12.1.3 互补性原则

互补性原则从人的角度看待自动化，它的目标是帮助人们长期高效地执行任务，而不仅仅关注短期效应。互补性原则观察有 / 无自动化的情况下，人的行为有什么变化。

在这种方法中，将考虑人们长期人工完成任务所能学到的知识，以及自动化所能改变的。例如，某人通过理解任务的主要目标和达成目标所需的基本功能，开始执行一项新的任务。随着时间的推移，这个人更多地理解了围绕任务的生态系统、异常情况以及一些全局性的目标，并根据这些知识改变自己的做法。

在自动化时，我们嵌入了目前已经积累的知识。这样做会抑制进一步的学习吗？这会不会改变这个环境的新手们的学习能力？人们的习惯会不会在较长的时期内因为自动化而变化？

互补性原则更多地采用认知系统工程（Cognitive Systems Engineering, CSE）方法，将自动化和人一起视为联合认知系统（Joint Cognitive System, JCS）。它考虑了人受到目标（主动）和事件（被动）驱动这一事实。联合认知系统的特征是在来自过程本身或者环境的破坏性影响下仍然控制局面的能力。该系统考虑了情况的动态特征，包括能力和需求可能随着时间推移或者在不同情况下发生变化的事实。

处理能力和需求变化的方法之一是考虑人和自动化之间的功能重叠，而不是任务的严格分离。这使得功能可以按照需要重新分布。人被视为系统中的活跃成分，是系统整体运作必不可少的高适应性、资源丰富的学习伙伴。

遗憾的是，使用互补性原则的自动化没有“银弹”或者容易使用的公式。但是如果你记得考虑人为因素和自动化对于系统运行者的长期影响，就更有可能设计出好的自动化系统。

12.1.4 系统管理自动化

在《A Model for Types and Levels of Human Interaction with Automation》(人与自动化系统交互的类型和水平模型)中, Parasuraman、Sheridan 和 Wickens 在 2000 年提出了一种自动化的实现途径。和 14.2.4 节中的讨论很类似, 他们观察到信息处理有 4 个阶段: 信息收集、信息分析、决策和采取行动。这些作者还观察到全人工和全自动之间的层次。自动化的不同级别包括: 系统提出建议供人类批准, 系统提出建议、在人类于一定时间内没有否决的情况下执行, 系统执行自己的决策并在事后通知人类, 以及计算机决定任何事项并自行采取行动、不需要或者要求人类输入。

Parasuraman 等人的结论是信息收集和分析阶段适合于高度的自动化。决策阶段应该采用对应于功能风险的自动化水平。低风险功能应该高度自动化, 高风险功能应该降低自动化水平。但是, 至于采取行动, 对高风险功能来说, 合适的自动化水平应该处于中等。

Allspaw (2012c) 的博客帖子《A Mature Role for Automation》(自动化的成熟任务)分为两部分, 详细地解释了 Parasuraman 等人的工作, 并将其与系统管理关联起来。他总结道, 更好的自动化模型是一种合作关系——互补性原则的应用。当自动化系统和运营人员协同工作时, 会和团队成员的工作一样提高彼此的价值。

团队合作的关键要素是信任和安全。Allspaw 引用了 Lee 和 See (2004) 建立可信自动化系统的方法:

- 为合适的信任而不是更高的信任度设计。
- 展示自动化系统过去的表现。
- 以运营人员能够理解的方式揭示中间结果, 展示自动化过程和算法。
- 简化自动化算法和运营, 使其更容易理解。
- 以和用户目标相关的角度, 说明自动化的目的、设计基础和应用程序的范围。
- 为运营人员提供预期可靠性、行为治理机制及预期使用方法的培训。
- 认真评估自动化系统的人格化问题, 如使用语音创建一个人造对话伙伴, 以确保合适的信任度。

12.1.5 经验教训总结

自动化的努力可能事与愿违。自动化通常被视为纯工程任务, 所以人为部分往往被完全忽略。系统以消除令人厌恶的乏味任务为目标, 使人可以应付更困难的任务, 这就将最难的部分留给了人, 因为这些部分过于复杂、无法自动化。这样, 消除了因为许多乏味任务的心理疲劳, 但是代之以负担更大的心理疲劳——需要持续地处理难题。

自动化可能给系统带来稳定性, 但是这种稳定性导致运营人员系统维护技能的缺失。紧急响应变得特别脆弱, 我们将在第 15 章讨论这一主题。

设计自动化时, 问自己一个问题: 这个自动化系统如何看待人为因素? 人是瓶颈、有害可变性的来源, 还是资源? 如果人是瓶颈, 能否在不影响人了解系统行为的情况下消除瓶颈,

能否不影响他们在必要的时候调整系统工作方式的能力？如果人是有害可变性的来源，你将约束环境和自动化的输入，使其更加可靠。对运行自动化系统的人有何影响？如何限制他们的工作？如何处理异常？如果人是资源，则需要人和自动化系统的更紧密耦合。但是在那种情况下，有两个令人苦恼的问题：“谁”完成“哪些”任务（分配）？谁掌控局面（责任）？

系统的长期运营可以分解为4个阶段：跟踪、调整、监控和目标选定。跟踪包括事件检测和响应输入或者检出事件的短期控制，自动化通常始于这一级别。调整涵盖长期控制，如状态间迁移的管理。12.3节中的案例研究是调整级别自动化的一个例子。监控涵盖长期控制、解释系统状态并选择计划保持预期参数。例如，容量规划、硬件评估与选择及类似任务就属于这个类别。目标选定根据企业整体目标设定系统的整体目标——例如，使用关键绩效指标（KPI，参见第19章）驱动预期行为。沿此链条上移时，较高级别的任务通常更适合于人，而非机器。当自动化推进到链条的上层，人和系统行为及其原因就更加失去联系，整个系统的长期健康可能受到危害。

自动化的隐含成本

“超级”自动化系统往往需要超级培训，这可能非常昂贵。员工招聘变得极其困难，开始限制公司以预期速度成长的能力，造成的机会流失成为沉重的代价。这种机会成本可能比系统节约的成本更高。此类困境是 Google 等公司发动极具侵略性的招聘活动、以雇用 SRE 的原因。

12.2 工具建设与自动化的对比

工具建设和自动化之间有所差别。工具建设（Tool Building）改进人工任务，使其更好地完成。自动化（Automation）寻求消除人为任务的需求。过程在不再需要人完成该过程时实现自动化，但是这并不会消除人的需求。一旦过程自动化，系统管理员的角色从完成任务变成维护自动化系统。

12.2.1 示例：汽车制造

运营自动化与汽车制造业中发生的变革类似。最初，为汽车车身外部金属板喷漆的工作是人工进行的。这一过程缓慢、精密且困难，需要很高的技巧。后来，大功率喷枪的发明改善了该过程。同样的人可以更好地完成工作，时间更短，浪费的油漆也更少。这种技术还降低了技能要求，从而降低了工作的进入门槛。但是，仍然有一个汽车面板油漆工的岗位。过程还没有自动化，但是有了更好的工具。

20世纪70年代，汽车制造厂自动化了汽车喷漆过程。它们部署了机器人喷漆系统，汽车面板油漆工的岗位被裁掉。员工现在维护机器人喷漆系统（自动化系统），这和金属面板喷漆是截然不同的工作。

12.2.2 示例：机器配置

在IT行业，我们正在进行类似的变革。在典型的云计算环境中，每台新机器必须为其在服务中的角色进行配置。人工过程包括加载操作系统、安装某些软件包、编辑配置文件、运行命令和启动服务。

系统管理员（SA）可以编写脚本完成这些任务。对每台新机器，SA运行这个脚本配置机器。这是对人工过程的改进，更快且更不易出错，机器的配置也更一致。但是，SA仍然需要运行脚本，所以设置新机器的过程还没有完全自动化。

自动化过程不需要系统管理员操作，或者将其操作减少到特殊情况的处理。继续我们的例子，自动化解决方案意味着，当机器引导时，发现自己的标识、配置且做好提供服务的准备。在大部分情况下，SA配置机器的任务被消除了。SA的角色转换为维护配置机器的自动化系统、处理不寻常的硬件或者操作系统。

12.2.3 示例：账户创建

云管理员常常维护组成服务交付平台的系统。为了向每个新开发人员提供访问权限，SA必须在多个系统上创建账户。SA可以人工创建账户，但是如果编写一个脚本在所有相关机器上创建账户，就能够节约许多时间，而且遗漏某个步骤的可能性也更低。每当有新开发人员加入公司，SA就使用这个账户创建工具。

如果SA编写一个定期运行的作业，检查人力资源数据库中是否列出新开发人员，然后自动创建新账户，就更好了。在这种情况下，SA不再创建账户，而由人力资源部完成。SA的工作是维护和改进账户创建自动化手段。

12.2.4 工具很好，自动化更好

大部分运营工作是重复的任务，如配置机器、创建账户、构建软件包、测试新发行版本、部署新发行版本、增加容量、服务故障切换、转移服务、转移或者减少容量。所有这些任务都可以用更好的工具改进，这些工具往往是自动化的跳板。

自动化的另一个好处是实现了有关不合格品（IT的术语是“故障”）的统计数字收集。如果某种情况容易使自动化失败，应该跟踪和调查。自动化往往不完整，某些边缘情况需要人工干预。那些情况也可以跟踪和分类，较为普遍的情况优先进行自动化。

工具建设很好，但是自动化是可伸缩云计算所必需的。

12.3 自动化的目标

在云服务中，自动化不是“锦上添花”，而是必备的——是实现增长所必需的。基于云的系统以许多方式成长：更多的机器、更多的子系统和服务、新的运营职责。如果必须为

每个服务或者每 N 台新机器雇佣新员工，基于云的服务就无法运作。没有一家公司能找到足够多的合格 SA，也无法承担这么多人的开支。考虑到越大的组织越难管理，人员管理问题本身就难以应付。

有一个流行的错误概念——自动化的目标是比人工更快地完成任务。这只是目标之一，其他目标如下：

- ❑ **有助于实现伸缩性。**自动化是劳动力倍增器，可使一个人作许多人的工作。
- ❑ **提高精确度。**自动化比人更不容易出错。自动化系统不会分心、不会失去兴趣，也不会随着时间的推移而变得粗心大意。随着时间的推移软件不断改进，以处理更多的边缘情况和错误情况。和硬件不同，软件会随时间推移而变得更强大 (Spolsky, 2004, 第 183 页)。
- ❑ **增强可重复性。**软件在执行任务时比人更一致。一致性是控制良好的环境的一部分。
- ❑ **提高可靠性。**一旦过程自动化，就更容易收集关于过程的统计数字和指标。这些数字可用于识别问题，提高可靠性。
- ❑ **节约时间。**要完成所有必要的任务，时间永远都不够。自动化任务需要花费的 SA 时间应该比人工任务少。
- ❑ **加快过程。**人工过程涉及思考和键盘输入，因此更慢。这两者都容易出错，更正错误往往要损失很多时间。
- ❑ **实现更多安全保障。**为自动化过程添加额外的事前事后检查很容易。这样做只需要一次性成本，但是改进了过程未来所有迭代的自动化。对人工过程添加更多检查会带来负担，从而诱使人们跳过这些检查。
- ❑ **用户权利下放。**自动化使非 SA 人员也可以完成某些任务。自动化使只有专家才能完成的任务变成最终用户可以使用自助服务工具完成的任务。在 12.2.3 节的例子中，来自人力资源部的人现在可以配置新账户，不需要系统管理员参与。任务委派可以节约时间和资源。
- ❑ **缩短用户等待时间。**人工过程只有在 SA 有空时才能进行。自动化系统可以在全天内运行，通常在 SA 有时间启动它之前就已经完成任务。即使自动化系统完成相同任务比人工慢，用户的净等待时间也可能更短。
- ❑ **缩短系统管理员等待时间。**许多人工过程包括完成一个步骤，然后等待一段时间才能继续下一个步骤，例如，等待新数据在系统中传播，或者等待机器重启。这类过程是所谓的充满“焦急和等待”步骤。如果等待时间很长，SA 可能在这段时间进行其他工作，但是这样做的效率往往很低。很多时候，我们因为开始从事别的工作而分心，忘记回到第一项任务或者忘记了背景。计算机比人更善于等待。

例如，构建软件包可能非常复杂。在用户输入一条命令时，好的工具会编译软件、

运行单元测试、构建软件包，可能还执行其他任务。而自动化应该消除运行工具的需求。Buildbot 是一个持续监控源代码存储库变化的系统。发生任何变化之后，该软件将检查、构建、打包和测试。如果所有测试通过，新软件包被放入一个存储库，可用于部署。最新的可工作软件包总是可用。许多软件包可能创建后从未用过，但是由于不需要 SA 投入精力，在有可用 CPU 周期的情况下没有附加成本。此外，自动化构建过程应该立即向开发人员反馈任何失败的自动测试，缺陷得以更快修复，得到更好的软件（参见第 9 章）。

另一个例子涉及新机器的配置。好的机器配置工具由系统管理员运行，可能有几个参数，如主机名和 IP 地址，结果是一台完全配置好的机器。而自动化适用于每台新安装的机器在目录或者外部数据库中查找自己的主机名、寻找自身功能的情况。此后，自动化系统配置机器的 OS、安装各种软件包、配置软件包并启动机器计划运行的服务。人工步骤被消除，机器自动投入运行。

有时候，自动化完全消除了某个过程的需求。典型的负载平衡器需要人工配置添加副本。许多工具可以使这种配置更容易、更不会出错。但是，全自动解决方案消除了不断重新配置的需求。我们已经看到许多系统中负载平衡器只配置一次，之后副本通知负载平衡器自身可用，由负载平衡器将它们加入循环轮次中。这种项目不自动化现有过程，而是通过发明新的运营方法，消除不断进行的重新配置。

案例研究：自动维修生命期

Google 使用 Ganeti 开源虚拟群集管理系统运行许多大型物理机器群集，这些机器为数以千计的用户提供虚拟机。物理机器很少出现故障，但是因为机器数量很大，硬件故障变得相当频繁。因此，SA 们花费许多时间应付硬件问题。Tom 参与了一个项目，自动化整个维修生命期。

首先，开发工具协助常见操作，这些操作都很复杂、容易出错，需要很高的专业水平：

- ❑ **排空工具**：在监控发现将要发生硬件问题的迹象（如可更正的磁盘或者 RAM 错误）时，所有虚拟机将迁移到其他物理机器。
- ❑ **恢复工具**：当物理机器意外死机，这个工具可以多次尝试开关电源。如果这些努力无法恢复机器，虚拟机将从另一台物理机器上的最后一个快照重启。
- ❑ **维修信息发送工具**：当机器需要物理维修，有一个规程通知数据中心技术人员，告诉他们哪些机器发生了问题，需要做什么。这个工具收集问题报告，使用机器维修 API 请求工作。报告中包含任何失效磁盘的序列号、失效 RAM 的内存槽号等。在大部分情况下，报告可以将技术人员导向准确的问题位置，缩短维修时间。

❑ **再吸收工具：**当机器从维修返回，必须评估、配置并重新加入群集。

这些工具都不断改进。很快，工具就比人更好地完成任务，可以比人完成更多的错误检查。值班任务只是简单地运行这些工具的组合。

现在，整个系统都可以通过组合这些工具全面自动化。构建一个系统跟踪机器状态（活动、有问题、维修中、再吸收中）。该系统使用监控系统的 API 和维修状态控制台创建触发器，在合适的时间激活合适的工具。结果是，值班任务从每天多次警报减少为每周 1~2 次警报。

来自自动化系统的日志用于驱动业务决策。通过加速移除已证明最不可靠的硬件的硬件型号，停机时间显著降低。

12.4 创建自动化系统

自动化有很多好处，但是需要投入时间和精力创建。

和任何编程一样，自动化最好在没有外界干扰的一段时间内创建。有时候，需要完成的其他工作太多，难以找到足够的时间专注于自动化系统的创建。你必须有意地投入时间创建自动化系统，不能寄希望于最终会有足够平静的时间。

在运行良好的团队中，团队的大部分时间应该花在自动化的创建和维护上，而不是人工任务。当人工任务开始成为工作负载的较大部分时，就应该采取措施，看看新的自动化手段能否恢复平衡。重要的是识别自动化在哪些方面有最大的作用，并首先加以处理。理解自动化不适用的地方也很重要。

12.4.1 为自动化投入时间

有时候，因为我们忙于紧急工作，而没有足够的时间去完成创建自动化之类的长期工作。打个比方，我们没有时间关闭漏水的龙头，因为所有的时间都花在拖地板上了。发生这种情况时，很难突破惯性，从根本上解决阻止我们拥有足够时间创建自动化的问题。

下面是为自动化工作投入时间的一些建议：

- ❑ 让管理层参与。经理们应该重新排定工作的优先级，强调自动化。
- ❑ 找到影响你完成重要任务能力的最大根源，并修复、缓解和消除这一根源。这可能意味着在你修理漏水的龙头时暂时忽略其他事情——即使这会造成一些故障。
- ❑ 消除，而不是自动化。找出可以消除的工作。例如，找出替其他人做的工作，并将责任归还给那个人。消除重复劳动。例如，如果你正在维护 10 个不同的 Linux 版本，最好的方法可能是将这个数字缩小到只有几个，甚至 1 个。
- ❑ 雇佣临时顾问以部署高级自动化框架，并培训人们使用方法。
- ❑ 雇佣初级人员（甚至是临时人员）完成“乏味的工作”，解放高级人员参加修复根本

问题的更大的项目。

- 从小处着手。小型任务的自动化可能有传染性。例如，使用 CFEngine 或者 Puppet 之类的配置管理工具自动化配置新机器的一个方面。一旦某项工作自动化，其他工作就容易得多了。不要试图用你所创建的第一个自动化系统解决所有问题。
- 不是更辛苦地工作，而是更好地管理时间。《Time Management for System Administrators》(Limoncelli, 2005) 有许多有益的建议。

简单就是美

Etsy 写了一个博客帖子，解释公司决定不采用新数据库技术的原因。问题不在于这种技术不好，而在于 Etsy 将因此维护两种不同的数据库软件系统、两种备份方法、两种测试过程、两种升级过程，等等。那样做的工作量太大了 (McKinley 2012)。

12.4.2 减少“苦活”

苦活是令人疲劳的体力劳动，与自动化的目标极端对立。但是，运营团队中往往会积累苦活，起先很少，逐渐增长。例如，可能有一个小项目因为高度地特殊性和不可重复而无法实现自动化。随着时间的推移，它重复出现，很快这一操作就变成巨大的负担。不久，团队就因为苦活而超载。

少量的苦活并不坏。实际上，这是正常的。不是所有任务都值得自动化。如果所有任务都完美地自动化，没有需要干的事情，通常意味着变化率和创新已经停滞。如果团队的总工时中有超过一半花在“苦活”上，应该是众所周知的危险信号。团队应该评估成员们花费时间的方式。通常有几个深入的“根源”问题，解决了它们就能够消除大量的人工劳动。建立直击这些问题根源的项目，搁置所有其他项目，直到再次实现 50=50 的平衡。

在 Google 这样的公司中，对这类情况的处理已经有确定的策略。具体地说，有一个正式过程，供团队宣布某件“苦活”为紧急情况。团队停下来考虑各种选择，制定计划，消除苦活的最大来源。管理层评估优先级排定计划，批准在实现平衡之前搁置其他项目。

系统管理员的时间非常宝贵，不能花在像人工任务这样线性回报的工作上。苦活导致人们精疲力竭，士气低落。通过减少苦活，我们不仅能够帮助公司，也能帮助自己。

12.4.3 决定自动化的首要任务

首先投入所有力量解决最大的瓶颈。可能有多个领域需要自动化，先选择影响最大的。

分析团队参与的工作和过程，找出最大瓶颈。瓶颈是未完成工作积压的地方。消除瓶颈能够提高吞吐率。

将所做的工作看作一条装配线。一个项目有许多步骤，一次在装配线上移动一步。这些步骤中的某一个将成为瓶颈。

在瓶颈上游所做的任何改进只会增加积压工作。在瓶颈之上改进是很有诱惑力的，这

往往很容易，或者是你所注意到的第一件事。有时候，解决这些问题会提高团队的效率，但是瓶颈深藏在其他团队的过程中。因此，你做出了改进，但是系统的总吞吐率没有改变，这样会造成瓶颈处等待的工作更多，造成更大的交通拥堵，使情况进一步恶化。

在瓶颈下游所做的改进无助于整体吞吐率。同样，这可能是最容易改进的地方，但是如果工作没有到达系统的这个部分，那么这部分的改进无助于整个系统。

瓶颈可能不涉及技术问题。可能是某个特定人员负担过重，或者整个团队淹没在紧急的小型任务而无法完成重要任务。

12.5 如何自动化

你不能自动化无法人工进行的任务。创建自动化的第一步是知道需要做的工作。首先人工完成过程，逐个步骤将完成的工作记入文档。按照文档重复过程，并随时更正。让其他人根据同一个文档操作，确保文档清晰。这是发现哪些步骤定义不清或者遗漏的最佳方法。

迈向自动化的下一步是原型化。创建工具独立完成各个步骤，确定每一步都是完备的且正确地工作。

现在，将单独工具组合为一个程序。成熟的自动化需要日志记录和所有调用的输入参数及返回值的完整性检查。每个阶段和最终版本都应该包含这些功能。自动化系统应该检查前一步的返回值，如果有问题，应该可以撤销前面的步骤。

下一阶段是制作全自动系统。识别导致 SA 运行工具的因素，代之以这些条件触发器的自动化。在实施最后的这一步之前，必须具备完整的日志记录和健全的错误处理能力。

在某些情况下，创建自助服务工具，以便非 SA 人员完成该任务，是很合适的。在这种情况下，安全性成为更大的问题。必须有一种权限模型，控制可以使用工具的人。必须进行检查，验证人们不会无意中做了坏事或者绕过系统获得访问特权。

12.6 语言工具

有许多产品、语言和系统可用于创建自动化，每个都有自己的优势和不足。这些工具涵盖了从基本到高级的各种能力范围。

12.6.1 Shell 脚本语言

Shell 语言为人们提供了在操作系统命令行提示符下输入的命令。很容易将 Bash 或者 PowerShell 提示符下交互式输入的命令序列转化为脚本，作为单一命令运行——可以人工运行，也可以由另一个过程（如 cron）运行。

Shell 脚本有许多好处。例如，它们可以实现很快的原型化。它们使程序员在非常高的

级别上工作，可以组合命令或者程序，组成更大、更强的程序。使用 Shell 脚本实现自动化也有缺点。最重要的是，Shell 脚本解决方案不具备其他语言的伸缩性，难以测试，不容易完成低级任务。

最常见的 UNIX Shell 是 Bourne Again Shell (Bash)，这是 Bourne Shell (UNIX /bin/sh 命令) 的超集。在 Microsoft Windows 世界里，PowerShell 是非常强有力的系统，简化了所有 Windows 系统和产品的自动化和协调。

12.6.2 脚本语言

脚本语言是设计用于快速开发的解释型语言，往往聚焦于系统编程。

常见的脚本例子包括 Perl、Python 和 Ruby。Perl 较旧，因为和 C 及 awk (UNIX 系统管理员传统上很了解的语言) 类似而很受系统管理员的欢迎。Python 的设计更清晰，代码也远比 Perl 容易理解。Ruby 在系统管理员社区中有许多追随者，在语法上与 Perl 和 Python 类似，但是增加了一些功能，更容易创建为特定任务而构建的小型语言。然后，可以用小型语言编写更多程序。

脚本语言比 Shell 脚本更灵活和全能。它们表达能力更强，可以更好地组织代码、更好地伸缩，鼓励面向对象编码和函数型编程等现代编程方法。它们还有更好的测试工具和更多预先编写的库。脚本语言的网络、存储和数据库访问能力胜过 Shell 脚本，而且有更好的错误检查。

Perl、Python 和 Ruby 都有大型模块库，这些模块执行文件操纵、日期和时间处理、HTTP 等协议的事务以及数据库访问等常见系统管理任务。你往往发现自己所编写的程序利用了许多库，可以将它们组合起来创建所需的功能。

脚本语言的缺点之一是它们的执行慢于编译型语言。这一缺点在近年来已经为更快的解释程序新技术所缓解。但是，由于前面描述的原因，速度在自动化中不总是很重要。因此，在速度受限于其他因素的系统管理工具中，语言的速度往往不是一个考虑因素。例如，如果程序总是等待磁盘 I/O，本身由快速还是慢速语言编写可能就没有关系了。

脚本语言的主要缺点是它们不适合于非常大的软件项目。虽然没有什么能够阻止你用 Perl、Python 或者 Ruby 编写很大的软件项目（许多项目已经这么做了），但是从逻辑上讲很困难。例如，这些语言不是强类型的。也就是说，变量的类型（整数、字符串或者对象）在使用变量之前不检查，使用的时候，这类语言将尝试进行正确的处理。例如，假定你连接一个字符串和一个数字：脚本语言将自动将数字转换为字符串，使其可以进行连接操作。但是，某些情况下脚本语言无法解决问题，程序将会崩溃。这些问题只能在运行时发现，而且往往存在于很少使用或者测试的代码中。相比之下，编译型语言在编译时检查类型，远远早于代码进入现场的时候。

因此，不建议将脚本语言用于包含数万行代码的项目。

12.6.3 编译型语言

编译型语言是大规模自动化的理想选择。以编译型语言编写的自动化系统通常比脚本语言编写的同样的系统伸缩性更好。

系统管理员常用的编译型语言包括 C、C++ 和 Go。如前所述，编译型语言通常采用静态类型，在编译时捕捉更多错误。

12.6.4 配置管理语言

配置管理（Configuration Management，CM）语言是专为系统管理任务创建的领域特定语言（Domain-Specific Language，DSL）。CM 系统是为维护机器配置所创建的，这些配置涵盖了从网络配置等低级设置到应该运行哪些服务及服务配置文件等高级设置的范围。

配置语言是声明性的。也就是说，程序员编写代码描述世界应该是什么样，语言理解需要什么样的变化才能实现那个目标。正如 10.1.3 节中的讨论，有些 CM 系统围绕收敛编排（将系统带到一个预期状态）设计，而其他则偏爱直接编排（遵循多步骤行动计划编排的能力）。

CFEngine 和 Puppet 是两个流行的 CM 系统。程序清单 12.1 和 12.2 分别说明了在 CFEngine 和 Puppet 中指定一个符号链接的方法。程序清单 12.3 和 12.4 展示了 Python 和 Perl 中的等价任务，这两种语言都是命令式语言（而非声明性语言）。注意，前两个程序清单简单地指定预期状态：链接名及其目标。相比之下，Python 和 Perl 版本必须检查链接是否已经存在，如果错误则进行更正，不存在则创建，并处理错误的情况。必须有很高的专业水平，才能知道这些边缘情况的存在、正确地处理并在各种条件下测试代码。这是很棘手的任务，难以完全正确处理。CM 语言简单地指定预期状态，利用了 CM 系统创建者已经拥有正确完成任务的知识和经验这一事实。

程序清单 12.1 在 CFEngine 中指定一个符号链接

```
files:
    "/tmp/link-to-motd"
    link_from => ln_s("/etc/motd");
```

程序清单 12.2 在 Puppet 中指定一个符号链接

```
file { ['/tmp/link-to-motd':
    ensure => 'link',
    target => '/etc/motd',
}]
```

程序清单 12.3 用 Python 创建符号链接

```
import os

def make_symlink(filename, target):
```

```

if os.path.lexists(filename):
    if os.path.islink(filename):
        if os.readlink(filename) == target:
            return
    os.unlink(filename)
os.symlink(target, filename)

make_symlink('/tmp/link-to-motd', '/etc/motd')

```

程序清单12.4 用Perl创建符号链接

```

sub make_symlink {
    my ($filename, $target) = ($_[0], $_[1]);
    if (-l $filename) {
        return if readlink($filename) eq $target;
        unlink $filename;
    } elsif (-e $filename) {
        unlink $filename;
    }
    symlink($target, $filename);
}

make_symlink('/tmp/link-to-motd', '/etc/motd');

```

配置管理系统通常有在其他定义基础上构建定义的功能。例如，可能有一个“通用服务器”的定义，包含了所有服务器必备的设置。另一个定义可能用于“Web 服务器”，继承了“通用服务器”的所有属性，并添加 Web 服务软件和其他属性。“博客服务器”可能继承“Web 服务器”的定义，并添加博客软件。相比之下，“图像服务器”可能继承“Web 服务器”属性，但是创建一个访问图像数据库并提供图像的机器，可能用更适合于提供静态图像的设置调整 Web 服务器。

通过构建这些定义或者“类”，可以积累一个非常灵活高效的库。例如，对“服务器”定义的更改自动影响从其继承的所有定义。

CM 系统的关键优势是 SA 可以在高级别上简洁地定义设置，而且很容易在共享定义和过程中保持最佳实践。主要的缺点是领域特定语言的学习曲线陡峭，从一开始就需要创建所有必要的定义。

12.7 软件工程工具和技术

自动化和任何其他软件开发项目类似，所以需要从所有现代化软件开发项目中得益的机制。你可能已经熟悉用于自动化的工具。尽管如此，我们仍然不断发现，运营团队没有使用这些工具，或者没有发挥出其全部潜力。

自动化工具及其支持工具（如缺陷跟踪程序和源代码存储库等）应该是供所有软件开发

参与者使用的集中化、共享服务。这种方法使协作变得更加容易。例如，如果所有团队使用相同的缺陷跟踪系统，在项目之间转移缺陷和其他问题就更容易。

服务交付平台和相关问题（如持续测试、构建和部署能力的需求）在第9章和第10章中已经讨论过。在本节中，我们讨论问题跟踪系统、版本控制系统、打包和测试驱动开发等技术。

12.7.1 问题跟踪系统

问题跟踪系统用于记录和管理缺陷报告和功能请求。每个缺陷报告和功能请求都应该通过该系统。这提高了问题的能见度，确保相关的工作出现在管理层生成的统计数字中，可能使其他人在你之前抽出时间来解决这个问题。通过一个系统管理所有工作也更容易避免重叠的工作。该系统使团队成员能够理解其他人正在做的工作，更容易向其他人移交任务（最后一点在你没有时间完成这些任务时特别有价值）。

不报告问题往往是种诱惑，因为报告问题需要花费时间，我们都是大忙人，或者我们认为“每个人都知道这个问题存在”，因此无须报告。这些都是需要报告的问题，不是每个人都知道这个问题存在，尤其是管理层。提高问题的能见度是解决问题的第一步。

缺陷与功能请求的对比

我们通俗地将问题跟踪系统称作“缺陷跟踪程序”，这实际上并不妥当。功能请求不是缺陷，需要采用不同方式跟踪。缺陷修复和新功能开发往往单独分配资源。这些工作流程常常也不一样。如果你选择不实施某项功能请求，问题也就迎刃而解，将其标识为“已解决”是合适的。如果选择不修复缺陷，缺陷仍然存在，这个问题仍然悬而未决，只能将其标识为“低优先级”。

问题跟踪系统通常可以配置为服务于不同团队或者项目，每个团队/项目都将其缺陷保存在自己的队列（Queue）中。可以选择为缺陷和功能建立不同队列（“运营-功能请求”和“运营-缺陷”），系统也可能有某种标记不同问题的手段。

将单据链接到子系统

确立一种方法，将单据关联到特定的子系统。例如，可以为每个子系统建立单独的队列，或者使用其他类型的分类/标记机制。做少量计划可以避免未来的麻烦。例如，想象有3个SRE团队，每个团队负责5个子系统。假定将要进行一次重组，改变各个团队负责的子系统。如果每个子系统有自己的队列或者分类，将单据转移到新团队很简单，只需要改变特定队列的所有者，或者用特定的类别标志移动所有单据。如果没有分类，重组就会很复杂，每个问题必须单独评估并转移到合适的团队。必须为所有当前问题进行这项工作，如果想要维护历史，还要对过去的所有问题进行这项工作。

确立问题命名标准

问题应该以统一方式命名。如果措辞清晰，理解和处理许多问题就会更容易。缺陷应

该说明其错误所在。如果一些缺陷以错误所在表达,其他则以功能应该如何工作表达,标题就会有歧义,令人困惑。例如,如果某人报告“帮助按钮链接到关于项目的页面”,这个句子描述的是需要修复的缺陷(应该链接到帮助页面)还是解释系统应该如何工作(在这种情况下,缺陷应该被关闭,标记为“似乎已经按照请求的方式工作”)就很不清晰。

功能请求应该从渴望新功能的人的角度描述。在敏捷技术中,推荐的模板是“作为[用户类型],我希望[某种目标],以便[某种理由]。”例如,请求可以这样开始:“作为SRE,我希望配置管理系统支持新机器类型‘ARM 64’,以便我们管理基于平板电脑的新Hadoop群集”,或者“作为用户,我希望能够通过API调用克隆虚拟机,以使用编程方式创建克隆。”

选择合适的问题跟踪软件

软件问题跟踪系统类似于IT技术支持单据系统。与此同时,它们两者之间有很大的差别,所以这两种功能需要不同的软件。问题跟踪系统的焦点是缺陷或者功能请求,而IT技术支持单据系统专注于用户。例如,在问题跟踪系统中,如果两个不同的人报告相同缺陷,这些报告被合并起来,或者第二个报告因为重复而被关闭。每个问题只存在一次。与此相反,IT技术支持单据系统是与用户沟通、帮助他们解决问题或者满足其请求的机制。如果两个人提交类似的请求,不会被合并,每个请求都和请求人一样是唯一的。

软件问题跟踪系统和IT技术支持单据系统在工作流上也不同。问题跟踪系统应该采用反映软件开发过程的工作流:接收缺陷、验证和修复;由不同的人验证修复;然后关闭问题。这涉及许多移交手续。必须生成的统计数字包括过程的每一步所花费的时间。IT技术支持的工作流更多的是和用户的通信往来。

12.7.2 版本控制系统

版本控制系统(Version Control System, VCS)是存储、访问和更新源代码的中央存储库。将所有源代码放在同一个位置,更容易协作和集中备份、构建过程等职能。VCS存储每个文件的历史,包括做过的所有更改。因此,可能可以看到特定日期的软件内容、撤销更改等。虽然版本控制系统最初是用于源代码控制的,但是VCS可以存储任何文件,而不仅是代码。

VCS框架都有类似的工作流程。假定你想要对系统进行更改,你“签出”(check out)源代码,将全部源代码复制到自己的本地目录,然后按照需要编辑。工作完成时,你“提交”或者“签入”(check in)更改。

旧的VCS框架在给定的时间内一次只允许一个人签出特定项目,避免人们做出相互冲突的更改。新的系统允许多个人签出同一项目。第一个签入更改的人比较轻松,其他人都要经历将自己的更改和过去的更改合并的过程。VCS软件协助合并,自动完成没有重叠的合并,并启动编辑器让用户人工合并剩下的内容。

分布式版本控制系统(Distributed Version Control System, DVCS)是一种新型的VCS。

在 DVCS 中，每个人都有完整的存储库。更改集在存储库之间传输。签出不仅提供特定版本的源代码拷贝，而且创建整个存储库的一个本地拷贝，包括完整的修订历史。可以将更改签入到本地存储库，独立于中央系统的变化。然后，在完成更改之后，合并一组更改提交到主存储库。这就使源代码控制变得民主了。在 DVCS 之前，你一次只能进行一项更改，更改必须经存储库控制人批准。在第一次更改被存储库所有者接受之前，难以继续下一次更改。在 DVCS 中，你是自有存储库的主人，和主存储库并行开发。对主存储库的更改可以受控方式拉取到你的存储库，只有在想要导入更改时才如此操作。在自己的存储库中所做的更改可以在需要时向上合并到主存储库，或者完全不合并。

VCS 不应该仅用于源代码。也就是说，配置文件也必须进行版本控制。当自动化或者工具的使用涉及配置文件更改时，应该自动化配置文件签出版本控制、修改并签入的步骤。不应该允许工具在 VCS 之外修改配置文件。

12.7.3 软件打包

软件开发之后，应该以包的形式分发。虽然这一主题在 9.6 节中已经讨论，但是我们在 这里再次提出该问题，因为维护服务交付平台的同一个运营人员往往临时分发自己的软件工具。

通过包分发自己的软件，可以利用所有保持软件最新的系统。没有这一能力，运营工具最终必须人工拷贝。结果是许多系统都过时，相应地带来所有问题。

理想状况下，运营工具包和其他软件一样，应该经过开发、Beta 测试和生产阶段（见第 9 章）。

12.7.4 风格指南

风格指南是指明源代码应该如何格式化，哪些语言特征应该鼓励、不鼓励或者禁止的标准。让所有代码遵循相同的风格，使代码的维护更容易，而且能提升代码质量。风格指南为质量和一致性设定了标杆，提升了整个团队的标准。风格指南对于缺乏经验的人还有指导作用。

程序员一般将大部分时间花在修改代码、为现有代码添加功能上，很少有机会从头开始编写新程序，并成为整个生命期的唯一维护者。当我们作为团队一起工作时，在第一次查看文件时就能够很快吸收其内容是至关重要的。这能够提高效率，如果编写遵循风格指南的代码，就能够获得这种效率。

风格指南基础

风格指南包含格式化和特征的建议。格式化建议包括规定如何使用缩进和空格。例如，大部分公司将缩进标准化为 4 个（或者比较不常见的 2 个）空格而不是使用 Tab 键，删除文件结尾的空行或者行末的空格，在函数之间放置一个空行，在大的段落或者类之前可能放置两个空行。

风格指南的规定不仅出于审美。它们还推荐特定的语言特征，往往直接禁止某些证明难以支持的语言特征。语言往往有两种实现某一功能的方法，风格指南选择其中的一种使用。语言还可能有一些容易出错或者因为其他原因而不便使用的特征，风格指南可能禁止或者不鼓励使用这些特征。

许多语言有自己的风格指南（Python 和 Puppet）。每个主流开放源码项目对其社区都有风格指南。大部分公司都有内部使用的风格指南。Google 运营许多开放源码项目，发布了自己的内部风格指南（除了修订版之外），用于十多种语言（<https://code.google.com/p/google-styleguide/>）。这使参与的社区成员可以保持当前的风格。Google 风格指南非常成熟，为创建自己的风格指南提供了很好的基础。

其他建议

风格指南中往往推荐特殊的标记法。例如，Google 风格指南推荐注释的特殊标记法。对临时、短期解决方案或者很好但尚未完善的代码使用 TODO 注释。使用 NB 注释解释不明显的决策。使用 FIXME 注释指出需要修复的代码并列出该问题的缺陷 ID。注释之后是编写者的用户名。图 12.2 展示了示例。

```
TODO(george): 下面的代码可以工作，但是应该重构以使用模板。
NB(matt): 这个算法不流行，但是可以得到正确的结果。
FIXME(kyle): 这将打断v3，2013-03-22修复。参见缺陷12345
```

图 12.2 Google 风格指南中的专用注释

正如 9.3.2 节中所提到的，源存储库可以调用程序在提交前验证文件。利用这些“提交前测试”调用风格检查程序，可阻止违反风格指南的文件提交。例如，对任何 Python 文件运行 PyLint，对任何 Puppet 文件运行 puppet-lint，即使是自制系统，如果 CHANGELOG 条目格式不完善，也会被拒收。结果是，不管团队成长到什么规模，都保持一致性。

12.7.5 测试驱动开发

测试驱动开发（Test-Driven Development, TDD）是一种软件工程方法，可以得到缺陷最少、可信度最高的代码。

传统软件测试方法是首先编写代码，然后编写测试代码的代码。TDD 采用相反的步骤，首先编写测试代码，因为代码还没有编写，这些测试会失败。然后，开发人员快速编写代码迭代并重新运行测试，这一过程持续到所有测试通过。那时，开发人员的工作完成。

例如，想象一下编写一个函数，取得 URL 并将其分为各个组成部分。该函数必须处理许多不同类型的 URL，这些 URL 可能嵌入或者不嵌入用户名及密码、使用不同的协议等。你将提出一系列 URL，作为代码应该能够处理的不同种类 URL 的示例。接下来，编写代码用每个示例 URL 调用函数，将结果与你认为应得的结果对比。正如 9.3.3 节和 10.2 节中所

看到的，这些测试称作单元测试。

编写测试代码之后，编写函数代码并逐步改进。定期运行测试，开始，只有涉及最基本 URL 的测试能够通过，然后，越来越多有趣的 URL 格式通过测试。当所有测试通过时，函数完成。

测试仍然保持而不删除。以后，你可以对代码进行大改，确信如果单元测试不失败，更改就应该没有意外的副作用。相反，如果需要改变代码行为，可以首先更新测试，预期不同的结果，然后修改代码直到这些测试再次通过。

所有现代化语言都有对应的系统，很容易列出测试并顺序运行。这使你可以专注于编写测试和代码，而不是管理测试本身。

12.7.6 代码评审

代码评审是在文件的任何更改提交到 VCS 之前，由至少一位其他人审核的过程。代码评审系统（Code Review System，CRS）提供一个用户界面，让评审人对文件中的特定行添加意见和反馈。原作者使用反馈更新文件。这一过程反复进行，直到所有评审人批准更改，集成到 VCS。

软件工程师对源代码进行评审，以获得“第二双眼睛”，找出错误并在代码加入主来源之前做出建议。这对配置文件也很有用。实际上，在配置文件保存在源存储库的环境中，代码评审是询问其他人在更改上想法的极佳途径。

CRS 还可以将任务委派给其他人，同时仍然可以提供质量控制，这就通过委派实现了批准权的扩展。例如，大部分公司不让任何人编辑他们的负载均衡器配置。但是，如果配置保存在 VCS 且要求负载均衡器团队中某个人的批准，负载均衡器更新的过程就变成自助服务。给出一个出色的“怎么做”文档，任何人都能编辑配置文件，向负载均衡器团队提交更改建议，后者检查之后提交激活更改。

使用代码评审系统还有其他的原因：

- ❑ **编写更好的代码。**代码评审不仅仅是风格检查，这是深入考虑代码，建议更好的算法和技术的时机。
- ❑ **双向学习。**不管是评审人还是被评审人更有经验，两者都可以提高技能、相互学习。
- ❑ **避免缺陷。**第二双眼睛会捕捉更多缺陷。
- ❑ **避免运行中断。**对配置文件使用 CRS 可以在生产环境中出现问题之前捕捉它们。
- ❑ **实施风格指南。**代码评审人可以反馈违反风格的情况，促使其改正。

花在代码评审上的时间应该与文件重要性成正比。如果更改是临时性的，对系统整体影响较小，审核应该全面但不应过度。如果文件处于重要系统的核心，有许多相互依赖，审核时应多加考虑。

健康团队的成员能很好地接受批评。但是，有一种技巧可以不让代码评审成为一种折

磨，或者不被视为折磨（这也同样重要）：批评代码，而不是人。例如，“这个算法很慢，应该用更快的算法替代”是正常的说法。“你不应该使用这个算法”就是对个人的指责。这种微妙的指责会削弱团队的凝聚力。团队成员应该有提出批评而不显得过分挑剔的能力，这种能力是在经理们以身作则和所有成员的监督下培养出来的。如果不这样做，就有毒害团队的危险。经理们如果不遵循这一建议，当团队成员在其他论坛上不公平地相互批评时也就不足为奇了。

12.7.7 编写刚好足够的代码

编写刚好能够满足功能需求或者修复缺陷的代码，不多也不少。

编写的代码太少意味着功能请求没有得到满足，也可能我们编写的代码过于简洁，难以理解。编写过多的代码造成维护的负担、造成更多缺陷，浪费时间。

编写太多的代码也是容易掉入的陷阱。我们可能编写比当前需要更复杂、更可配置的代码，或者添加认为某一天会需要的功能。我们把这种方法叫作“面向未来”，并且指出这将在未来需要该功能时节约我们的时间。创建超越需求是有趣和令人兴奋的。

在现实中，历史教育我们，我们并不擅长预测未来。我们认为未来需要的功能中有80%是错误的。过多的代码需要花费时间开发，在一个每个人都抱怨没有足够时间完成工作的行业中，我们不应该造成额外的工作。额外的代码对于未来的代码维护者来说也是负担。因为无用代码而增大20%的程序，维护的难度要增大20%以上。现在，编写较少的代码会在将来节约整个团队的时间。大部分编码都涉及现有代码的维护和功能增加，很少从头开始一个新项目。考虑到这一事实，使维护更容易才是关键。

无用的代码更容易包含缺陷，因为它们没有进行过演练或者测试。如果测试这些代码，我们就要花费时间增加测试、自动化测试系统就要做更多的事情，未来的维护者会因为所有自动化测试“由于某种原因而必须存在”而无法确定是否可以删除某个功能。最终需要这个功能时，可能已经过了几个月或者几年，由于需求和环境已经改变，将会发现新的缺陷。

话虽如此，有一些面向未来的工作是合理的。主机名或者文件名等常量应该可以通过命令行标志或者配置设置。将重要函数或者类分开到不同的库，以便可以供未来的程序重用。

我们已经发现有助于抵制“面向未来”诱惑的3个技巧。首先，使用测试驱动开发，迫使自己在所有测试通过时停止编码。其次，添加TODO()注释，列出你想添加的功能，往往会降低实际编写代码的情感需求。最后，风格指南应该明确地阻止过度面向未来，鼓励主动删除已经过时的无用代码或者功能。这将确立可应用于代码评审时的高标准。

12.8 多租户系统

多租户是一个系统为多组客户提供服务，每组客户严格区分，以免受其他组操作影响的情况。

找出所有其他团队能够从中得益的自动化方法，比仅为自己的团队创建自动化系统更好。可以在内部提供软件发行版本，使其他团队可以运行类似的系统。更好的做法是，可以创建自己的系统，使其同时服务其他团队。

设计系统的配置管理特性时，可以通过**多租户**（multitenancy）实现更大的灵活性和委派管理。这意味着设置系统，允许单独的组（租户）控制他们自己的代码库。

多租户框架的重要特性是可以供多个组以自助方式使用，每个组的使用和其他组隔离。这样做需要一个权限模型，每个团队不受其他团队更改的影响。即使只向所有团队提供一个服务，每个团队也是自己的安全域。

案例研究：多租户 Puppet

Google 有一个集中化的 Puppet 服务器系统，提供对使用该系统的单独团队的多租户访问。Mac 团队、Ubuntu 团队、Ganeti 团队和其他团队都可以管理自己的配置，而不会相互干扰。

该系统非常精密，为每个租户的文件提供全面源代码控制的区域，以及用于开发、测试和生产的单独预演环境。添加到集中化系统的任何功能有益于每个人。例如，Puppet 团队所做的任何使服务器处理更大量客户的工作，都有利于所有租户，当 Puppet 团队实现了从企业防火墙之外安全访问 Puppet 服务器时，所有团队都得到了即使在移动中也能够保持所有机器更新的能力。

虽然 Google 的系统使每个租户都可以自助方式工作，但是保护措施仍然存在，任何团队都不能修改其他团队的文件。Puppet 清单（程序）以根方式运行，可以更改所运行机器上的任何文件。因此，重要的是（例如）Ubuntu 团队不能更改 Mac 团队的文件，反之亦然。这样做本质上使 Ubuntu 团队可以访问所有 Mac 机器。这通过一个简单而强大的权限系统实现。

12.9 小结

系统管理员的大部分工作应该专注于 SA 任务的自动化。云计算系统管理员的目标应该是在人工运营工作上花费的时间少于一半。

工具建设优化系统管理员所做的工作，是通往自动化的一个重要步骤。自动化意味着用软件代替人工完成任务，这些软件往往和某个人合作。自动化始于记入文档、定义完整的可重复过程。文档可作为工具用于创建使过程更快速、更可靠的脚本。最终，通过创建自助服务工具或者通过数据库或者配置文件中的变化自动触发的过程，完全自动化该任务。自助服务工具特别有用，因为它使其他团队自给自足。当某项任务实现自动化，SA 的任务从完成任务变成维护完成任务的自动化系统。更好的思路是将自动化视为运营部门的合作伙伴而不是替代。

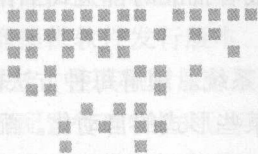
自动化有许多级别,从完全人工到不需要人否决或者批准的自主式工作系统。根据任务的类别和风险,有不同的适用级别。

SA 可以选择脚本语言或者编译型语言创建自动化系统。理解每种方法的好处和不足,就能为工作选择合适的工具。配置管理系统也有益于某些形式的自动化。配置管理工具使用声明性语法,以便指定最终结果。CM 工具可以理解如何将系统带入该状态。

最佳实践是将用于自动化的所有代码和配置文件置于版本控制之下或者跟踪更改的数据中。和软件开发环境中一样,自动化工具不应该在生产环境中开发或者测试。相反,应该在专用的开发环境中开发,然后在测试区域预演以进行质量保证测试,再投入生产环境。

练习

1. 列出自动化的 5 种好处。哪一种适用于你自己的环境,为什么?
2. 记录当前环境中需要自动化的一个过程的文档。记住,文档应该包含完成过程所需的每个步骤。
3. 用决策矩阵描述如何排定自动化的优先级。
4. 何时应该使用脚本语言代替编译型语言,为什么?在哪些情况下你会使用编译型语言进行自动化?
5. 想象你将在当前环境中实施配置管理系统。你会选择哪一种,为什么?
6. 你的时间有多少花在一次性任务和未经自动化的运营工作上?如何优化?
7. 列出自动化可能带来的 3 种挑战。对每一种挑战,描述所能采取的措施。



设计文档

案例研究：多租户 Puppet

Google 有一个集中化的 Puppet 服务器系统，提供对使用该系统的单独团队的多租户访问。Mac OS、Ubuntu 和其他操作系统都可以管理自己的配置。该系统非常精密，为每个租户的文件系统提供完全控制的区域，以及用于测试和生产环境的部署环境。添加新租户非常简单，只需在 Puppet 服务器上添加一个新的配置即可。

真诚；简短；在座位上坐好。

——富兰克林 D. 罗斯福

在本章中，我们介绍设计文档，讨论其用途和相关的最佳实践。设计文档是所提出和完成项目的书面描述，规模可大可小。设计文档是项目的路线图，记录了你的成就。

13.1 设计文档概述

设计文档是提出或者完成项目的描述。它们记录目标、设计、考虑的替代方法以及其他细节，如成本、时间表和公司策略的相容性。

写出将要完成的工作，迫使你认真思考细节，也就能迫使你做出计划。当文档成为沟通思路的手段，它就能促成协作性的设计。书面文档意味着团队成员们更不容易因为某些细节而大吃一惊，帮助你在下一步行动之前获得共识。在项目完成之后，设计文档可以作为记录工作的工件和团队的参考文献。

好的设计文档深入所提项目或者更改的细节，包括有关选择原因的信息。例如，文档可能包含算法的详细描述、特殊的打包参数和二进制文件安装位置，并解释配置文件为什么放在 /etc 目录中，而不是放在其他地方。关于命名空间选择（如服务器/机架名称的命名空间设计）的设计文档不仅描述命名结构，还给出了选择命名空间的原因，以及命名空间与现有命名空间是否集成的背景信息。

文档本身有助于在项目上达成一致。有时候，“显而易见”的更改并不真的那么明显，人们会找出错误或者提出问题。正如 Linus Torvalds 所说，“众人使缺陷无处藏身。”

案例研究：Amazon 的逆向作业

Amazon 鼓励工程师们从描述客户看到的景象开始，然后逆向构建设计。工程师们首先撰写发布产品的新闻稿。他们设计描述客户如何从产品获益的营销材料，识别产品功能并回答常见问题。这就发展出了将被创建产品的愿景。只有在这项工作结束之后，所创建的设计才能实现愿景。这一过程在《Black》(2009)中做了全面的描述。

13.1.1 记录更改和依据

设计文档也可用于描述更改而不是项目。简短的设计文档格式很适合于在小更改上达成一致，如新的路由器配置、采用配置管理系统新功能的计划或者文件层次结构的新命名方案。

可以创建设计文档捕捉提议更改的细节，作为团队意见的“共鸣板”。如果有正式的更改控制过程，这个文档可以包含在更改请求中。

13.1.2 作为过去决策存储库的文档

存档的设计文档可以成为参考文献，供需要理解项目工作原理以及达成目标过程的任何人使用。如果有可供学习的文档，新的团队成员就能很快地跟上。现有团队成员也会发现，当他们需要快速回顾特定项目时，文档很有用处。与其他团队协作时，让他们阅读文档，比临时解释更实用，也更专业。

当知识的传递不需要和专家私下互动时，团队的效率更高。团队可以快速成长，其他人可以更快地采用我们的服务，我们也可以自由地在团队之间转移，因为知识并没有锁定在某个人身上。设计文档所传达的不仅是设计，还包括目标和灵感。由于人们可以长时间参考设计文档，它们是保存背景和语境的很好场所。

团队往往在电子邮件存档中保留决策历史。问题在于，电子邮件本身是隐藏在某人的邮箱中的。团队的新成员无法找到这些电子邮件，它们也不会出现在文档存档搜索结果中。加上一封简单的邮件经过一年时间的反复更改，团队基本上不可能访问整组信息。

不要只是说，将一切写下来。规程、设计和智慧只有写下来才能体现价值。除非将你的意见写下来，否则其他人无法参考。在这个意义上，电子邮件就像谈话，而不是写作(Hickstein 2007)。

13.2 设计文档剖析

设计文档分为许多部分。首先是最高级别的信息，随着文档的深入而逐步细化。我们将通过标准化格式、标题和这些部分的顺序，帮助读者理解。

附录 D 的 D.2 节中包含了完整设计文档的一个例子。D.1 节中有一个模板样本。

设计文档的各个部分是：

- ❑ **标题：**文档标题。
- ❑ **日期：**最后修订日期。
- ❑ **作者 / 审核人 / 批准人：**审核人是按照请求对文档内容提出反馈的人。批准人是必须提供文档审批意见的人。
- ❑ **修订号：**文档应该有和软件发行版本号类似的修订号。
- ❑ **状态：**状态可以为草稿、审核中、已批准或者进行中。有些组织采用更少或者更多类别。
- ❑ **摘要：**2-4 句的项目摘要，包括项目的主要目标和达到目标的手段。
- ❑ **目标（或者“范围”）：**项目所要实现的目标，通常采用项目列表方式。内容包括无形的过程目标，如标准化或者指标达成。重要的是考察自己的目标并仔细核对，确保设计文档记录每一个目标。在某些组织中，每个目标都有编号，在整个文档中用编号引用目标，每个设计描述引用对应的目标。
- ❑ **非目标（或者“范围之外”）：**通常使用一个项目列表提供这一信息。非目标列表应该明确指出本项目的范围不包含哪些。这个部分能够阻止“本项目不解决 XYZ 问题”之类的审核意见，因为我们已经在此说明，本项目的意图不是解决“XYZ 问题”。
- ❑ **背景：**读者为理解设计所应知道的典型信息。本部分可能提供到目前为止的简短历史，标识所使用的任何缩写词或者不寻常的术语。记录对本项目产生限制或者约束的过往决策，例如“我们公司的策略是尽可能只使用开源软件，这会阻止我们在本项目中使用商业化解解决方案 XYZ。”
- ❑ **概要设计：**设计的简短概述，包括概要的工作方式。应该描述设计原则，但是不一定描述实现细节。
- ❑ **详细设计：**完整的设计，包括框图、配置文件样板、算法等。这将是项目计划实现目标的完整和详细描述。
- ❑ **考虑的替代方案：**被拒绝的替代方案列表，以及拒绝的原因。有些此类信息可能已经包含在“背景”部分，但是在一个特殊的位置加入它可以平息批评。
- ❑ **特殊约束：**有关安全性、审计控制、隐私等方面的特殊约束列表。这有时是可选的。将公司过程的各个部分记入文档，如架构审核、相容性检查和类似活动。可以在这一部分中罗列所有约束，也可以将每个约束包含在单独的小节中。公司项目审核的任何强制过程都应该在模板中有独立的小节。

文档可能有更多的部分，有些是可选的。下面是模板中可能有用的部分：

- ❑ **成本预测：**项目成本——初始资金和运营成本，加上保持系统运行的成本预测。
- ❑ **支持需求：**运营维护需求，这和成本预测相关，因为支持人员有工资福利成本，硬件和许可证也有成本等。

- ❑ **时间表**：项目事件发生的时间表，以及相互之间的关系。
- ❑ **安全性考虑因素**：和项目相关的特殊安全性问题，如数据保护。
- ❑ **隐私与 PII 考虑因素**：和用户隐私或者匿名性相关的特殊问题，包括根据适用规则和监管处理个人可识别信息 (PII) 的计划。
- ❑ **依从性细节**：为符合 SOX、HIPPA、PCI、FISMA 或类似法规监管的依从性和审计计划。
- ❑ **启动细节**：试运行或者启动运营细节和需求。

13.3 模板

为人们提供一个模板，可以为他们创建自己的设计文档提供指导，这比步进式指导或者样板文档更好。模板应该包含最终文档应该有的所有标题，甚至包括可选的标题。模板可以用每部分预期内容的描述作为注解，还可以包含有益的提示和技巧。这些注解应该采用不同的字体或者颜色，通常在用户填写模板时被删除。

附录 D 的 D.1 节中可以找到一个模板示例，将它作为组织模板的基础。

模板应该很容易找到。在引入模板时，每个人应该得到通知，知道模板的存在，并且得到模板所在位置的链接。该链接应该出现在其他位置，例如设计文档存档的目录或者人们经常查看的另一个位置。确保模板出现在内部网络搜索中。

模板应该以用户将使用的所有格式或者他们喜欢的格式出现。如果人们使用 MS-Word、HTML、MarkDown（一种 Wiki 格式）或者 OpenOffice/LibreOffice，就提供所有这些格式的模板。仅以你最喜欢的格式提供，假定其他人可以凑合使用，他们就不容易采用你的系统。

提供一个短模板和一个长模板也很有用。短模板可以用于初始提案和小的项目。长模板可用于复杂项目或者正式提案。

模板还应该易用。每当发现某人不使用模板，就是调试模板问题的时机。以非对抗的方式询问他们为什么不使用模板。例如，告诉他们，你希望得到如何使更多人采用模板的反馈，并询问需要做什么，让他们更容易使用模板。如果他们说找不到模板，询问他们搜索的位置，确保更新这些位置，包含链接。

13.4 文档存档

应该有保存所有设计文档的单一存储库。通常，这可以是简单的文档列表，每个列表项链接到文档。为了使文档存档有益于读者，人们应该很容易找到所需的文档。简单的搜索机制可能有用，但是如果有一个页面列出所有文档标题，且该页面本身可以搜索，人们通常会很高兴。

向文档存档中添加设计文档应该很容易。添加或者更新存档的过程应该是自助服务。如果该过程包含向某人发送电子邮件，那个人就会不可避免地成为瓶颈。这种障碍将阻止人们添加新文档。

最简单的存档创建方法是使用 Wiki。利用这种方法，人们在主页上添加他们的文档链接。之后，真正的文档保存在 Wiki 或者其他地方。人们通常添加到已经存在的任何结构中，因此，在制作主页时投入时间和思考，为其他人制作简单、易于维护的主页是很重要的。

维护存储库的另一种方法是为人们创建一个添加文档的位置——例如，一个源存储库或者文件服务器上的子目录——并添加自动生成索引的脚本。创建一个文件命名系统，以便自动组织文档。最简单的方法是为每个团队设置一个独特的前缀，让团队自行选择前缀。生成索引需要可解析以提取数据的文档格式，或者供人们列出索引所需数据的标准化位置，例如，和文档放在一起的 YAML 或者 JSON 文件。然后，脚本生成所有文档的索引，列出文件名、标题和最近修订号。

13.5 审核工作流程

应该为设计文档的批准制定流程。批准过程不需要高开销的活动，只需要简单地回复包含设计文档的电子邮件。人们必须知道得到批准不需要巨大的工作负荷，否则他们就不会寻求批准。

组织设计文档的批准流程有很多种。其中大部分流程的风格可以归纳为 3 种：

- ❑ **简单：**创建一个草稿，四处转发以征求意见，在必要时修订，不需要正式批准。
- ❑ **非正式：**类似于简单工作流程，但是需要来自项目批准人或者外部评审委员会的审核。评审委员会提出反馈意见，并对过去重复出现的错误提出警告，而不提供认可意见。
- ❑ **正式工作流程：**多个批准阶段，包括项目批准人和外部评审委员会。典型的例子包括防火墙或者安全性审核、项目预算审核和发行计划审核。每个审核阶段都记录所有意见，改变修订号或者日期轨迹。

不同类型的文档需要不同的批准流程。非正式文档可能使用轻量级流程或者没有批准过程。根据文档的范围或者影响，可能需要较大规模的批准。你的批准工作流可能由更改控制或者依从性需求决定。

13.5.1 审核人和批准人

设计文档通常列出作者、审核人和批准人。作者是贡献文档的人。审核人是提供反馈的人。批准人是提供批准进入下一步的人。

审核人通常包含可能进行实际项目的团队成员，还可能包含主题专家以及作者希望得

到意见的任何人。增加某人为审核人，是得到受项目影响的另一群体中“参考意见”的好办法。你应该期望得到审核人的有意义的意见，包括需要对文档做出更改的意见。

批准人通常包含依赖项目结果的内部客户，和需要签发项目所用资源的经理。有时候，批准人包含公司强制的其他审核和控制过程——例如，隐私评估或者项目所用 PII 的审核。执行 SOX、HIPAA 或者 PCI 等规章依从性审计的人员也是设计文档过程中的批准人。批准人通常发表是/否的意见，只有做出“否”的决定时才做出解释。批准必须全体一致，还是多数批准意见可以压倒一个或者多个不批准的意见，取决于单独过程。

13.5.2 获得签字同意

审核人和批准人之间的区别在于批准人可以阻止项目。试图在所有人中取得一致会妨碍所有进程，尤其是需要取得不直接受项目成功影响的人们同意的项目。在项目边缘的人不应该拥有否决权。每个人都有意见，但不是每个人都有投票权。审核人/批准人的区分澄清了每个人的角色。

高专业性的多重批准

Google 的设计文档批准过程包括来自管理不同集中化服务的组织签字同意。例如，如果项目向集中化日志存档发送数据，设计文档必须包含“日志”部分，详细说明每个事务产生的日志数据字节数，并且证明存储和轮转策略正确。这种项目需要中心日志团队签字同意。对于安全性、隐私和其他批准人，需要类似的部分。

如果评审委员会在文档中发布搜索的项目检查列表，与其协作就更容易些。这样，作者可以更好地准备评审。发布评审委员会检查列表是对同事时间和你的时间的尊重，公开检查列表之后，有缺陷的文档也会变得更少。评审委员会可能是一组批准人或者审核人（或者两者的组合）——在准备文档时预先确定这一点，最符合你的利益。

13.6 采用设计文档

在组织中实施某种设计文档标准可能是一个难题，因为这需要文化的改变。人的变化很慢。要让人们更容易采纳新的行为方式，必须以身作则，并得到管理层的支持。

提供模板是推动这种策略采用的第一要务。确保模板容易访问，并以人们最想使用的格式出现。

为想要在其他那里看到的行为建立模型。在任何合适的地方使用设计文档模板，并严格遵循其格式。人们会模仿他们看见的成功人士表现出的行为。直接和组织中受人尊重的成员对话，希望他们成为早期采用者。他们愿意或者不愿意采用设计文档标准的反馈具有启发性。

应该回报早期采用者，并强调他们取得的成功。这并不一定要采用大奖赛的形式，可

以简单地在每周内部会议上发表一些正面的意见。即使他们对模板的使用并不完美，也要加以称赞，这是回报他们愿意踏出第一步。将如何更正确地使用模板的建议留到一对一的反馈中。获得管理层支持是另一条途径。如果你是一位经理，就可以使设计文档成为自顶向下的要求，坚持让团队使用它们。另一种开始使用设计文档的途径是使它们成为某些批准过程的要求，如预算审核、安全性或者防火墙审核委员会或者更改控制。不要将盲目地坚持模板格式作为批准的要求。内容比形式更重要。

当团队成员发现模板很容易使用，就会开始在其他项目中采用。

你自己也要使用模板，如果模板对你个人没有好处，对其他人就可能没有好处。使用它还有助于发现问题，这些问题很容易通过更新模板来解决。当 Tom 加入 Stack Exchange 时，他所做的第一件事是创建一个设计文档模板并将其放在部门的 Wiki 上。他在提及这个模板之前将其用于两个提案，每次使用时都修订和完善模板。在使用几次之后，他才向队友们提及模板，很快其他人也使用该模板。

13.7 小结

设计文档可以作为项目路线图和记录，它的主要功能是描述项目，还可以向团队传达项目的关键特征（谁、什么、为什么、何时和如何）。设计文档对更改控制和项目规格也有作用。好的设计文档不仅捕捉所做的决策，还包括决策的依据。

采用设计文档可能为团队带来文化上的挑战。首先将出色的模板放在容易找到的位置上，使其以团队使用的文档格式出现。为设计文档创建一个中央存储库，按照最近更新的顺序组织。简化存储库主目录页面中添加设计文档的操作。

批准是良好设计文档文化的一部分。创建文档草稿并且征求团队某些成员的意见。对于简单的工作流程，电子邮件足以传递文档供其他人审核。对于更正式的工作流程，可能需要强制性的评审委员会或者多个批准步骤。在所有工作流程中，每次在文档中添加审核意见时都要更新设计文档的版本号。理想状态下，文档存储库的主页也应该在修订发生时更新。

模板应该包含谁、什么、为什么、何时和如何等重点信息。对小型或者非正式项目可以使用较短的模板，对于复杂或者正式的项目使用更大、更复杂的模板。模板至少应该有标题、日期、作者、修订号、状态、摘要、目标、概要设计和详细设计。可以添加替代或者可选部分，根据组织需要定制模板。

练习

1. 设计文档的主要功能是什么？
2. 列出好的设计文档管理系统的特质。

3. 你的组织目前有没有标准化的设计文档格式？如果没有，在组织中实施标准需要什么条件？
4. 你的组织中使用何种强制的设计评审过程？哪些利益相关方必须担任审核人或者批准人？
5. 描述组织的设计文档等价物，以及和本章介绍的模型相比的优势和弱点。
6. 为组织创建设计文档模板。向同事展示并获得关于格式以及易用性的反馈。
7. 用设计文档格式为提议或者现有项目撰写设计。
8. 用设计文档格式重写组织使用的现有文档。
9. 在某些蠢事或者乐事上使用设计文档格式——例如，公司野餐的设计或者快速致富的计划。

14.1.1 从SLA开始

在设计组织的值班方案时，从服务的SLA入手，反向推导出能够造成符合服务SLA的值班SLA。然后，设计满足值班SLA的值班方案。例如，假定某个服务的SLA允许在2个小时的停机时间之后才开始累计罚款。假定典型的问题可以在30分钟内解决。极端的问题下需要花费30分钟进行系统切换，但是通常有尝试其他解决方案30分钟之后才进行这一步。这意味着从运行中断开始到问题得到积极处理之间的时间必须小于1个小时。

值班方案必须考虑到服务中断对客户造成的影响。对于某些服务，中断可能会导致严重的后果。例如，如果服务中断导致客户无法访问其银行账户，那么这可能会导致客户的不满和损失。因此，值班方案必须考虑到服务的可用性和可靠性。此外，值班方案还必须考虑到值班人员的技能和经验。值班人员必须能够处理各种类型的问题，并且能够在压力下保持冷静。最后，值班方案还必须考虑到值班人员的福利和补偿。值班人员通常需要在非工作时间工作，因此他们应该得到相应的补偿和福利。

14.1.2 值班人员花名册

花名册(Roster)是轮流值班的人员名单。这个名单由具有资格的员工组成，通常由经理和运营人员组成。所有运营人员都应该在花名册上。这通常被认为是所有运营团队成员的“值班表”。花名册通常由经理制定，并且会根据需要进行更新。花名册通常包括值班人员的姓名、职位、技能和经验。此外，花名册还可能包括值班人员的联系方式和紧急联系人。花名册的目的是确保在需要时有足够的人员来值班，并且确保值班人员具备必要的技能和经验。

随时待命

小心……这世界需要更多的警惕。

——伍迪·艾伦

值班是处理异常情况的手段。即使我们尝试自动化所有运营任务，也总会有一些职责和边缘情况无法自动化。这些异常情况可能发生在一天中的任何时候，它们不会刚好安排在朝九晚五的时段里。

简单地讲，异常情况就是运行中断和任何不予干预就会导致运行中断的情况。更确切地说，是服务违反（或者将要违反）SLA 的情况。

运营团队需要一个策略，确保异常情况立刻得到干预，并采取合适的措施。应该设计策略，减少未来此类情况的发生。

最佳的策略是确定一个时间表，任何时刻都至少有一个人将这类问题的处理作为首要职责。在值班轮次期间，值班人员应该保持联络畅通，并且在需要采取行动的计算机和其余设施的范围内。在两次异常现象之间，值班人员应该专注于与班次中面对的异常现象相关的后续工作。

在本章中，我们将讨论这个基本策略，以及许多变种。

14.1 设计值班

值班（Oncall）是让一组人员轮流负责异常情况（更常被称作紧急情况，“警报”是听上去比较不令人畏惧的说法）。值班安排通常提供 24×7 的覆盖。通过轮班，人们可以从这种压力巨大的职责中解脱出来、恢复正常生活和休假。

接收到警报时,值班人员做出响应,解决问题,使用任何必要手段避免违反SLA,包括不能长期解决问题的捷径。如果他们不能解决问题,有一个升级系统让其他人介入。在问题得到控制之后,任何后续工作应该在常规上班时间内进行——特别是根源分析、事后剖析和长期解决方案的设计。

正常情况下,任何时候都有一个人被指定为“值班人员”。如果有来自监控系统的警报,这个人接收警报并管理问题直到解决。在上班时间,这个人正常工作,但总是从事很容易中断的项目。在下班之后,这个值班人员应该距离计算机足够近,以便快速反应。

还需要一个策略,处理值班人员无法到达的情况。这可能是因为通勤、网络中断、紧急的健康状况或者其他问题。通常指定一位辅助值班人员,当主值班人员在一定时间之后尚无反应的情况下介入。

14.1.1 从SLA开始

在设计组织的值班方案时,从服务的SLA入手,反向推导出能够造成符合服务SLA的值班SLA。然后,设计满足值班SLA的值班方案。例如,假定某个服务的SLA允许在2个小时的停机时间之后才开始累计罚金。假定典型的问题可以在30分钟内解决,极端的问题下需要花费30分钟进行系统切换,但是通常在尝试其他解决方案30分钟之后才进行这一步。这意味着从运行中断开始到问题得到积极处理之间的时间必须小于1个小时。

在这一个小时中,必须发生如下情况。首先,监控系统必须检测到运行中断。如果每5分钟轮询一次,在3次尝试之后才发出警报,至多15分钟某位值班人员就应该收到警报。最坏的情况是假定最后一次正常轮询正好发生在运行中断之前。我们假定警报每5分钟发送一次,直到某人做出反应;每3次警报导致从主值班人员升级到辅助值班人员,或者从辅助值班人员升级到整个团队。最糟糕的情况(假定团队没有接收到警报)是6次警报——30分钟。从接收到警报开始,值班人员可能需要5~10分钟登录系统、开始工作。目前为止,在实现“把手放到键盘上”之前,运行中断已经持续了50~55分钟。考虑到我们估计最多需要60分钟才能解决问题,现在留给我们的时间只有5分钟了。

每个服务都不同,所以必须分别计算。如果管理许多服务,应该根据所需的响应时间创建几类服务来简化过程:5分钟、15分钟、30分钟及更长。可以定义每类服务的监控、警报和补偿方案,并对所有新服务重用,而不是每次都从头来过。

14.1.2 值班人员花名册

花名册(Roster)是轮流值班的人员名单。这个名单由具有资格的运营人员、开发人员和经理们组成。所有运营人员都应该在花名册上。这通常被认为是所有运营团队成员的职责的一部分。

运营人员刚加入团队时,他们的训练计划应该以快速熟悉值班所需技能为重点。训练的一部分应该是跟随值班人员工作。在大部分公司中,新雇员应该在3~6个月里就能够处

理值班任务，但是这一时间各不相同。

有些组织不要求高级运营人员分担值班任务。这并不明智，会使高级运营人员处于和所负责的运营现实脱节的风险中。

开发人员也应该在花名册上。这能帮助他们了解自己所开发系统中固有的运营问题，指导他们在未来的设计决策。换言之，当第2章中所列出的功能缺失会影响他们时，对增加这些功能的阻力也就消失了。这会激励对运营问题的修复，而不是提供变通方法。

例如，如果因为某个问题，值班人员在凌晨3点被叫醒，他可能使用某种变通手段，并提交缺陷报告以请求更固定的解决方案，这一解决方案可以避免值班人员未来在不正常的时段内被叫醒。如果开发人员不参与值班，这种缺陷很容易被忽略。如果他们在值班人员花名册上，就有修复这些缺陷的动机。

运营和开发人员的工作优先级保持一致很重要，因为如果做不到这一点，运营问题将不会得到应有的重视。在过去的紧缩套装软件时代中，人们接受开发人员与运营需求脱节的情况，但是这种时代已经一去不复返了（如果你不知道什么是紧缩套装软件，去问问你的爷爷奶奶）。

技术经理、团队领导和技术项目经理也应该分担值班任务。这使他们和运营的现实保持接触，帮助他们成为更好的管理者。我们还发现，在更广泛的人才使用值班剧本和工具时，它们会在精确度和效率上保持更高的标准。换言之，如果你认真地编写剧本，以至于技术经理可以遵循剧本工作，那么这个剧本就应该是很不错的（致阅读本书的经理们：我们的意思是，在人们发现你将看到他们的工作成果时，就会保持更高的标准）。

14.1.3 值日

值日（Onduty）和值班很类似，但是主要专注于处理用户的非紧急请求。值日与值班不同。

许多运营团队也有来自用户的请求队列，管理方式类似于IT技术支持通过服务台自动化工具接收和跟踪请求的方式。值日是确保总是有一个人处理这些单据的职能。否则，单据可能在一段时间内被忽略。值日职能的俗称很多：单据时间、单据周、单据任务或者值日。通常，值日人员花名册上的人员和值班人员花名册上的一样。

值日人员的主要职责是在SLA范围内响应服务单据——例如，初始响应是1个工作日，夜间、周末和假日不提供服务。通常，值日人员负责筛选新单据、排定优先级、处理大部分单据并将特殊情况委派给合适的人员。

大部分组织都有不同的值班和值日安排。在某些组织中，值班人员自动成为值日人员，这简化了时间安排，但是有个缺点：如果发生紧急情况，服务单据将被忽略。话虽如此，在出现大的紧急情况时，无论如何都会忽略服务单据。而且，如果因为紧急情况而在深夜叫醒值班人员，第二天这个人可能不在最佳状态，期望这个人在如此条件下响应服务单据也不是好主意。因此，除非单据负荷极轻，否则应该尝试使用单独的值班安排表。

有些组织完全没有值日安排。它们可能处于没有用户会向其提交单据的情况。有些团队感觉接受单据不符合 DevOps 的思想。他们自己是协作者，而不是接受服务请求的服务台。如果有协作请求，应该通过常规业务渠道处理。在协作中，分配的任何工作都应该和其他项目相关任务一样进入工作跟踪系统。

14.1.4 值班表设计

值班表的结构有许多变种。每个人的值班轮次长度应该以对团队意义最大的方式安排。下面是常用的一些变种：

- **每周**：每人一次值班一周。下一班次从每周的同一天开始，如每周三正午。在周中换班比周末换班更好。如果换班发生在周三，每个值班人都有一个完整周末的旅游或者其他娱乐受到限制。如果在周六换班，则会毁掉两个值班人员连续两周的周末。在周一，可能从周末积压了很多后续工作。最好是让处理周末警报的人员完成后续工作，或者至少在值班时间内生成相应的单据和文档。这可以实现清晰的移交，使每个人在值班任务结束之后尽快返回项目工作。
- **每天**：每个人一次值班一天。这似乎比每周的安排更好，因为班次没有那么长，但是这意味着值班的频率高了很多。每周的安排可能意味着每六周值班一周。对于小团队，每天安排可能意味着每六天就要值班一次，再也没有完整的一周可以休假了。
- **日内倒班**：在一天内有多人值班，每个人负责每天或者一个班次的不同时段。例如，两班倒的安排每天有两个 12 个小时的班次。一个人从早上 9 点到晚上 9 点，另一个人则负责夜班。这样，如果班次中发生警报，仍然有人可以睡觉。三班倒的安排是每人 8 小时：上午 9 点到下午 5 点，下午 5 点到凌晨 1 点，凌晨 1 点到上午 9 点。
- **跟随太阳**：运营团队的成员生活在不同时区内，每个人在正常清醒的时段（白天）值班。如果团队位于加利福尼亚和都柏林，在加州时间早上 10 点和晚上 10 点换班意味着所有成员都在上班时间和睡觉的时间内当值，并且有足够的重叠时间供团队间沟通。分散到多个时区的团队可以每天有 3 个或者 4 个班次。

还可能有许多种变种。有些团队喜欢用半周代替一周。跟随太阳可以根据人们所在位置采用 2、3 或者 4 个班次。

警报频率

有这么多种值班安排方案，很难决定使用哪一个。可以开发一个策略，由警报的频率决定轮班的频率和时长。人们在不超出负荷且不需要连续几天保持清醒的情况下状态最佳。值班系统如果能使每个警报都接受因果分析等后续工作，就能够改进运营。在某人再次值班之前，这个人应该有足够的时间完成所有后续工作。

例如，如果警报极其稀少，每周少于一次，每人一次值班 7 天是合理的。如果每天有 3 次警报，2~3 次轮班可以让人们在警报和后续任务之间有时间休息。如果后续工作规模很

大，可能需要分为两个班次，每班半周。

如果警报频率很高，需要更复杂的方案。例如，在某些安排中，由两人值班：如果一个人忙碌，则由另一个人接收警报，另外还有两名辅助人员作为备份。有些系统按照地理位置划分工作，在世界上的每个重要地区都采用不同的轮班方案和时间安排。

时间表协调

值班时间安排可以和其他时间表协调。例如，可能有一条政策：在当值轮次的前一周，你要参加值日。升级点可能有专用的时间表。例如，深入了解某个特定服务的人可能需要协调，确保他们不会同时休假。那样确保如果值班时需要将问题升级到这些人，就有人员可用。有时候，升级时间安排是非正式的共识，有时候是包含 SLA 的正式安排。

补偿

补偿会驱动时间表的某些设计要素。在某些国家，如果响应时间低于某一时间间隔，必须向值班人员提供补偿。正常上班时间之外的值班补偿通常是每小时为正常日工资的 1/3。补偿可能以现金或者补休的形式发放。如果值班人员被召来参与行动，补偿比例可能不同。人力资源部门应该提供所有必要的细节。在某些国家，值班补偿没有法律规定，但是好的公司无论如何都会提供，否则就是不道德的。“跟随太阳”方案的好处之一是可以最大化在某个位置的正常上班时间的值班时间，最小化预算内的额外补偿金额。

14.1.5 值班日程表

值班日程表 (Oncall Calendar) 记录“谁”在“什么时候”值班。将时间表和花名册的理论变为具体的细节。监控系统利用这一信息决定向谁发送警报。

提前足够的时间制定日程表，可以预先规划所有的考虑因素——假期、旅行和其他职责。3~6 个月通常就足够了。制定日程表的细节根据团队而有所不同。由 6 人组成，每个周三换班的团队可以简单地使用在线日程表的“重复事件”功能安排每个人的值班时间。冲突和其他问题可以在成员之间解决。

对于更复杂的时间安排和较大的团队，相应地需要更复杂的日程表构造策略。这种系统使用共享的在线数据表，如 Google Drive。数据表单元格表示下三个月的每个时间槽。由于团队规模很大，每个人都应该有 3 个时间槽。该系统是“先来先服务”的，有许多非正式的讨论可以在不同人之间交换时间槽。这种协商持续到某个截止点为止，此时时间表锁定。这一系统对在制定时间表时恰好外出的人不太公平。

有些公司采用较为规则的方法。由于存在许多内部和外部服务，Google 每个月需要制定数百个单独的日程表，在有人编写程序为他们完成这项任务之前，每个团队都花费很多时间协商和组合日程表。为了使用这个系统，团队创建一个 Google 日历，每个人插入事件，标记哪些日子里他们完全没有空闲、空闲但不愿意值班、空闲或者愿意值班。这一系统使用一个配置文件，描述每个班次的长度、某人进入下一轮次之前是否需要时隙等参数。

然后，系统从 Google 日历中读取人们的偏好并合成数据，直到创建合理的值班日程表。

14.1.6 值班频率

当班的频率需要小心考虑。每个警告都需要一定数量的后续工作，应该在下一班次开始之前完成。每个人还应该在值班轮次之间有足够的进行项目工作，而不仅仅是后续工作。

来自某个警报的后续工作可能很多。撰写事后剖析报告可能是费力的任务。根源分析可能涉及大量调查，需要持续几天甚至几周。

班次越长，接收的警报越多，值班人员尝试同时进行的后续项目也就越多，这可能超出其负荷。

班次之间的间隔越短，工作就越可能无法在下一班次开始之前完成。

同时进行一两项事后剖析是合理的，但是太多就不可能完成了。因此，班次时间应该足够长，以积累一两次重大警报。根据服务的不同，时长可能是一天、每天 8 小时的一周或者 24×7 服务的一周。如果希望这个人完成两项事后剖析、进行项目工作并且可能临时休假，下一个班次应该至少相隔 3 周。如果某个服务接收太多警报而无法实现上述方案，这个服务就有更深层次的问题。

值班轮次可能造成很大的压力，如果压力的来源是每个班次太过忙碌，可以考虑较短的班次，或者指定第二个值班人员以处理“溢出”的事务。如果压力的来源是人们对处理警报的能力不自信，建议进行更多的培训。培训团队使其更加轻松处理运行中断情况的方法在第 15 章中讨论。

14.1.7 通知类型

监控和其他服务需要引起运营人员注意的紧急情况有许多个级别。只有最紧急的情况需要发出警报。每个紧急程度级别应该有自己的通信方法。如果紧急警报只是简单地发送到某人的电子邮件收件箱，他们可能不会及时注意到。如果通过短信向值班人员发送非紧急信息，就会出现“狼来了”的情况。

最佳的选择是建立一个非常高级的分类系统：

- **警告值班人员：**违反 SLA，或者条件检测发现不干预就会造成 SLA 违反的情况。
- **创建单据：**应该在一个工作日内引起注意的问题。
- **登记到一个文件：**不需要运营人员注意的情况。我们不想丢失信息，但是不需要得到通知。
- **什么也不做：**没有有用的信息。不应该发送任何消息。

在某些组织中，所有这些情况都通过电子邮件传达给整个系统管理员团队。在这些情况下，所有团队成员不得不将所有消息过滤到一个文件夹，将其忽略。这就违背了发送这些消息的初衷。

电子邮件很可能是最差的警报机制。期待某人坐下来监视电子邮件收件箱是愚蠢的，而且是对每个人时间的浪费。使用这种策略，人们如果离开或者参与其他项目就无法看到新的警报。

报告状态检查结果的每日邮件也是一个坏主意。如果状态正常，在日志中记录这一事实。如果发现问题，自动打开一个单据。这可以避免多名运营人员无意中同时处理同一问题。如果问题很紧急，应该立刻向某人发出警报，那么为什么每天只检查一次？这种情况下，应该频繁地向监控系统报告状态，正常地发出警报。

但是，使用电子邮件作为辅助机制是一个好主意。也就是说，向传呼机发送信息或者创建一个单据时，还通过电子邮件接收一份拷贝是很有用的。许多系统有订阅此类消息、允许精确过滤的机制。例如，通常可能配置单据系统，发送特定队列中新单据或者已更新单据的通知邮件。

一旦触发警报，有许多途径通知值班人员。最常见的警报方法如下：

- ❑ **单向和双向传呼机**：通过地面广播接收文本消息的手持设备。双向传呼机可以发送一个响应，确认消息已经收到。
- ❑ **SMS（移动电话文字消息）**：向个人移动电话发送文本或者 SMS 消息很方便，因为大部分人都有手机。在某些国家，传呼机明显比 SMS 可靠。在其他国家则相反。如果为一个国家的同事创建警报系统，不要假设对你有效的方案在另一个地方也可行。当地人员应该测试两种方案。
- ❑ **智能手机小应用**：智能手机小应用（App）可以显示比短消息更多的信息。但是，它们往往依赖于互联网连接性，这不总是可用。
- ❑ **语音电话**：使用语音合成器和其他软件呼叫值班人员电话并与之交谈，要求按下一个按钮确认消息（否则，将激活升级列表）。
- ❑ **聊天室机器人**：聊天室机器人是一个软件机器人，“坐”在团队的聊天室中公布任何警报。这是保持整个团队参与并在必要的时候做好帮助值班人员的准备的一种实用手段。
- ❑ **警报仪表盘**：警报仪表盘是一个显示已发送警报历史的网页，提供实用的上下文信息。
- ❑ **电子邮件**：电子邮件绝不应该是值班人员接收警报的唯一方式。坐在电脑边监视收件箱是糟糕的时间使用方法。但是，用邮件将每个警报发送给值班人员，作为备份方法是很有用的。这样，可以接收到完整的消息，SMS 会将消息截断为 160 个字符。

至少应该使用两种方法确保消息传送。传呼服务可能停机。睡着的值班人员可能需要 SMS 铃声和语音电话才能叫醒。应该有一种方法能够将完整、未删节的消息发送到稳定的存储媒体。电子邮件很适合于这一用途。最后，还应该部署聊天室机器人和其他方法，特别是因为这种策略使整个团队都能了解问题的发生。这些手段一开始似乎只是新奇的玩意，

但是很快就会成为不可或缺的工具。

14.1.8 下班时间维护协调

有时候,维护必须在下班时间完成。例如,另一个团队可能进行需要你参与的维护,如进行服务故障切换,使其不会受到计划内停机的影响,或者验证状态等。

将这些任务分配给当时正在值班的人员是很常见的。对于大部分情况来说,这都不成问题。但是,这可能造成值班人员同时处理警报和协调不相关的问题。

如果值班人员投入这些任务,辅助值班人员应该处理维护窗口内发生的所有警报。也可以将维护窗口协调任务指派给辅助值班人员。

14.2 当值

我们现在已经知道如何开发花名册、时间表和日程表,可以考虑当值时相关的职责了。值班人员在每一班次之前、之中和之后都有不同的职责。

14.2.1 当班前的职责

在班次开始之前,你应该确保做好准备。大部分团队有**班次准备检查列表**(Oncall Shift Preparedness Checklist)。检查列表上的项目验证可达性和访问能力。可达性指的是警报通知系统正常工作。你可以向自己发送一个测试警报,验证是否一切正常。访问能力意味着你能够在响应警报时访问所需的资源:VPN 软件正常工作,便携电脑的电池充好电,保持执行操作所需的冷静等。

改正发现的任何问题都需要时间。因此,检查列表应该相应地提早激活,为协商延长当前值班人员班次或者找到替代人员留出足够的时间。例如,发现你的双因子身份验证设备不能正常工作时,可能需要时间设置新设备。

14.2.2 常规值班职责

一旦班次开始,你应该……实际上也没什么特别的。在工作时间内,你应该和平常一样,但是只承担可以在必要时打断的任务。如果参与会议,从一开始就警告参会人员你正在当班、随时可能必须离开,是一个好主意。如果在正常工作时间之外当班,应该在必要时休息,基本保持正常的生活,但是必须保持联系。如果需要一段路程,如下班回家,在这一过程中从辅助值班人员中确定临时替代。

在当班期间不应该做的是坐在计算机旁边,监视服务仪表盘和监控系统,看看是否能发现问题。这是浪费时间,本来那就是监控系统的作用。如果这类活动是值班的要求,那么值班系统的设计就很糟糕了。

有些团队有每个班次完成的**任务清单**。这些任务的例子包括验证监控系统正常工作、

检查备份的运行、检查内部使用软件相关的安全警报。这些任务应该通过自动化消除。但是，在自动化之前，将这些任务分配给当值人员是在团队中分摊任务的方便手段。这些任务通常可以在警报之间完成，并且认为可以在班次中的某个时候进行，但是明智的做法是及早进行，以免遗忘。不过，如果班次开始于某位值班人员的睡觉时间，期望这些任务在班次一开始就进行是不合理的。因为非紧急情况叫醒值班人员，不是一种健康的做法。

任务可以每天、每周或者每月进行。在所有情况下，都应该有一种登记任务完成的机制。维护一个共享数据表，由人们标记完成的事项，或者自动打开一个单据，在任务完成时关闭。所有任务都应该伴随一个缺陷 ID，申请通过自动化或者其他手段消除任务。例如，可以通过建立对监控系统进行监控的系统，自动验证监控系统正在运行（参见 16.5 节）。当临时散热系统最终被替换时，清空临时散热设备冷凝水桶的任务应该被消除。

在没有活动警报时，值班应该是相对无压力的工作。

14.2.3 警报职责

一旦接收警报，你的职责就变了。现在，你负责验证问题、修复问题并确保后续工作完成。你可能不是完成所有工作的人，但是负责通过委派和移交，确保所有工作的进行。

你应该确认警报处于前述的 SLA 范围内。确认警报可使警报系统知道，不应该向升级列表中的下一个联系人发出警报。

快速修复与长期修复的对比

现在，问题已经着手解决。你的首要任务是提出最佳解决方案，在 SLA 范围内解决问题。有时候，我们可以在长期修复和快速修复之间选择。长期修复将解决根本问题，避免未来发生同样的问题，这可能包括编写代码或者发行新软件，很少能够在 SLA 范围内进行。快速修复在 SLA 范围内解决问题，但是可能只是推迟问题的发生。例如，重启机器可能解决眼下的问题，但是几天之内可能需要再次重启，因为技术问题并没有得到解决。不过，重启现在就可以完成，避免违反 SLA。

一般来说，鼓励偏向于长期修复而不是快速修复：“一针不补，十针难缝。”但是，值班不同于常规的工程工作，快速比长期完善更加优先。因此，必须淘汰无法适应 SLA 的解决方案，快速修复可能是唯一的选择。

请求帮助

在必要时请求帮助也是值班人员的职责。升级到更有经验或者博学的人，或者在问题发生的时间足够长时，找到比你精力更充沛的人。你没有必要只手补天，可以请求其他同事帮助。特别是在运行中断规模很大或者同时发生多个警报时，可以联系其他人，不一定要自己解决问题。你的职责是确保问题修复，有时候，将问题交给合适的人并进行协调是最好的做法，而不是尝试自己处理一切。

后续工作

解决问题之后，优先考虑的因素变成升级问题能见度，以便完成长期修复和优化。对于简单的问题，提交缺陷报告或者在现有报告上加注解可能就足够了。较为复杂的问题需要编写事后剖析报告，捕捉发生的情况，对未来如何避免做出建议。这样，我们可以建立一个反馈循环，确保运营逐步变好，而不是越来越糟。如果问题没有可见性，核心的问题就不会得到解决。不要以为“每个人都知道出了问题”就意味着问题会得到解决。并不是每个人都知道出了问题。不能期望负责决定哪些项目优先得到资源的经理们无所不知，或者可以读到你的心思。提交缺陷报告就像捡垃圾：你可能以为其他人会这么做，但是如果每个人都无所作为，环境就永远不会干净。

知道问题的根源之后，应该对警报进行分类，以便生成指标。这有助于发现趋势，结果信息应该用于确定未来的项目优先级。以可搜索的方式记录涉及的机器也很有用。未来的警报可以和过去的警报关联，发现诸如同一台机器重复出现故障等简单趋势。其他后续工作在 14.3 节中讨论。

14.2.4 观察、确认、决策、行动

观察、确认、决策、行动（OODA）循环是 John Boyd 为作战行动开发的，其设计是用于喷气战斗机空战之类的情况，因此适用于需要快速反应的高压力环境。Kyle Brandt（2014）将 OODA 应用到系统管理，推广了这一思路。

假定警报与表示网站缓慢且经常超时的指标相关。首先我们要观察（Observe）：检查日志、读取 I/O 计量指标等等。

接下来，我们确认（Orient）自己的方向。确认是分析和解释数据的行动。例如，日志包含许多字段，但是为了将数据转换为信息，必须查询日志找出异常情况或者模式。在这一过程中，我们根据数据和经验提出假设，找出真正的根源。

现在，我们要做出某种决策（Decide）。有时候，我们确定需要更多信息并开始收集。例如，如果有迹象表明数据库速度很慢，就要从数据库收集更具体的诊断信息，并重新启动循环。

最后一个阶段是采取行动（Act）并做出改变：解决问题、测试假设或者提供更多数据进行分析。如果确定某些查询使数据库服务器变得很慢，最终必须有人采取行动修复问题。

OODA 循环几乎总是有许多次迭代。较有经验的系统管理员可以符合逻辑、快速而顺畅地循环迭代。而且，随着时间的推移，好的团队会开发工具加速循环，并且更好地协作，巩固这一循环。

14.2.5 值班剧本

理想的情况下，系统可能生成的每个警报都和描述响应的文档相匹配。值班剧本

(On-call Playbook) 就是这个文档。剧本的一般格式是需要检查或者完成的事项列表，如果到达列表的末尾，问题升级到值班升级点（本身可能是人员的轮换）。这就创建了一个自动修建的反馈循环。如果人们感觉到在深夜将其叫醒的升级情况太多，就会通过改善文档，使值班人员更加自给自足，改正这一问题。

如果人们觉得编写文档不重要，或者是“其他人的工作”，他们就会因为没有创建合适的检查列表，导致值班人员在夜间任何时间将他们叫醒。觉得编写文档是低等工作的人在午夜被叫醒之后突然感受到编写文档的快乐，是令人印象深刻的事情。这种反馈循环的结果是每个检查列表的详细程度都达到了合适的平衡。

编写值班剧本时，确定每个检查列表的详细程度可能是一个挑战。“检查数据库的状态”对于有经验的人来说可能是足够的。但还应该包含所需的实际步骤，以及对正常情况的说明。对每个基本信息（如怎样登录）都详加解释通常是没有必要的。

有目的地选择假定的知识基准。应该有相应文档帮助新员工提升到那个水平。然后，值班文档可以假定员工达到了基准水平，这有助于避免文档过于详细和重复的情况。要求作者包含太多的细节，可能成为编写文档的障碍。

14.2.6 第三方升级

有时候，升级必须包含运营团队之外的人，如不同服务或者供应商的值班团队。升级到第三方有特殊的考虑因素。

在运行中断期间，不应该浪费时间研究与第三方联络的方法。所有关联的第三方都应该记入文档。应该有单一的全局可访问列表，列出每个内部团队的值班信息。如果值班人员的数量在每一班次中都会变化，该列表应该用当前信息动态生成。

供应商联络信息表通常包含应该和团队之外的人共享的信息，因此存储的方式不同。对于每个相关单位，这个专用列表应该包含供应商名称、联络信息和开启一个支持问题所需的任何信息。例如，可能包含许可证信息或者支持协议号。

供应商升级管理技巧

有时候，供应商支持问题多日之后尚未解决。此时：

- ☐ 应该有一个人在问题发生期间与供应商接口。否则，沟通和语境都会变得毫无组织。
- ☐ 信任，但是要验证。在验证问题解决之前，不能让供应商关闭问题。
- ☐ 只在没有得到结果时才向销售联络人报告问题。否则，销售代表可能加以干涉，从而惹恼技术支持人员。
- ☐ 接管问题，不要假设供应商会这样做。跟进问题，确保过程的推动。
- ☐ 在可能的时候，尽早打电话跟踪。这可以驱使供应商整天都忙于你的问题。

14.2.7 班次结束时的职责

你的班次最终会结束，这时候要将值班任务移交给下一个人。确保过渡的顺利，将问题的背景交给下一个人，保证未决的问题不会被遗忘。

策略之一是编写**班次结束报告**（End-of-Shift Report），用电子邮件发送给值班花名册上的所有人。只发给下一个值班人很容易出错。你可能错误地选择了不合适的人，或者不知道有替班的情况。将报告发送给每个人，使整个团队都了解最新的情况，为每个人提供在必要时参与的机会。

班次结束报告应该包含发生的任何重要事件，以及下一班次必须知道的任何事项。例如，应该说明持续的运行中断，或者需要人工监控的行为。

另一个策略是**明确移交**（Explicit Handoff）到下一班次。这意味着，上一当班人必须明确地陈述，他将把职责以交给下一当班人，下一当班人必须正面地确认移交。如果下一当班人没有或者无法确认移交，责任仍然在原当班人身上。这一方法用于可用性需求很严格的场合。

在这种情况下，班次结束报告可以是口头的，或者通过电子邮件发送。书面的报告更好，因为可以传达给整个团队。

如果班次的轮换发生在一方或者双方可能在休息的时候且没有持续的问题，简单的电子邮件就够了。如果有持续的问题，下一值班人应该接收到警报。如果上一当班人在班次轮换时正在休息，常见的做法是发出**临时班次结束报告**（Provisional End-of-Shift Report），注明除非现在和班次轮换之间有警报，否则这一报告可以认为是最终的报告。

14.3 两次当值之间

两次当值之间的正常工作时间应该花在项目工作上，包括与值班时的警报相关的后续工作。虽然这些工作可以在值班时间完成，但是休息通常更重要。

和值班相关联的项目包括无法在值班期间实施的长期解决方案，以及事后剖析报告。

14.3.1 长期修复

每个警报都可能产生无法在值班期间完成的后续工作。大的问题可能需要因果分析，以确定运行中断的根源。

在前一个例子中，重启机器解决了一个问题。因果分析可能说明，软件存在内存泄漏的情况。和开发人员一起寻找和修复内存泄漏，是长期的解决方案。即使你不是能够最终修复代码的开发人员，除了鼓励提供最终解决方案的开发人员之外，还有许多工作可做。你可以进行监控收集关于问题的信息，以便进行事前和事后比较。可以和开发人员一起理解问题对业务目标（如可用性）的影响。

14.3.2 事后剖析

事后剖析是分析运行中断，将发生的情况和原因记入文档，并对如何在未来避免这种运行中断提出建议的过程。

好的事后剖析过程向管理层的上游和下游传达信息，向用户传达正在采取措施的信息；向同等的团队传递信息，以便了解问题的相互作用（好的和不好的）；还传达给不关联的团队，使其能从问题中吸取教训。

事后剖析过程不应该在运行中断结束之前开始，不应该使人们从运行中断修复中分心。

事后剖析是持续改进策略的一部分。每次用户可见的运行中断或者 SLA 违反之后都应该进行事后剖析，并以实施事后剖析报告中的建议作为结束。这样，我们就可以将运行中断转变为学习，将学习所得转变为行动。

事后剖析的目的

事后剖析不是相互指责，其目标是识别出现问题的地方，以便在未来改进过程，而不是确定归咎于谁。没有人应该为被炒鱿鱼或者自己与技术失误相关联而感到恐惧。相互指责会影响开放性，而这是获得必要透明度，识别问题并作出改进的关键。如果事后剖析活动是“指名道姓的批评”过程，工程师们对于未来的行动细节和观点就会三缄其口，“掩盖罪责”（Cover Your Ass, CYA）的行为成为常规，信息流变得更少，管理层也就更不知道工作的进行情况，其他工程师对系统中的陷阱更没有多少认识。结果是，发生更多的运行中断，继续这一循环。

对于自身的行为造成运行中断的工程师来说，事后剖析过程记录了他们当时所采取行动的详细描述、观察到的效果、他们的预期、所做的假设以及对事件发生时间轴的理解。工程师们应该可以在不担心惩罚的情况下完成这个过程。

这并不是说团队成员们和错误毫无关系，他们和许多事情联系在一起，现在他们是负责培训整个组织如何不再未来犯相同错误的专家，应该推动与改进当前情况相关联的设计工作。

当责（Accountability）文化（而不是指责）能够培育重视创新的组织。如果用指责来逃避责任，整个团队都会蒙受损失。关于这一主题的更多信息，建议阅读 Allspaw（2009）的文章《Blameless Postmortems and a Just Culture》（无指责的事后剖析和公正的文化）。

每个高优先级警报的事后剖析报告

在 Google，许多团队都有在监控系统向值班人员发送信息时撰写事后剖析报告的策略。这是为了确保没有任何问题被忽略或者“扫到地毯下面”。因此，Google 对正常运行事件的高标准没有出现任何下滑，而且，这还使警报系统得到了极好的调整，生成的虚假警报非常少。

事后剖析报告

事后剖析报告包含 4 个主要部分：运行中断的描述、事件时间轴、促成条件分析（Contributing Condition Analysis, CCA）和避免未来运行中断的建议。运行中断描述应该说明被影响的对象（例如，内部客户或者外部客户）以及受到干扰的服务。事件时间轴可以在事后重现，但是应该清晰地确定真正发生的事件顺序和时间。CCA 应该详细说明运行中断的原因，并且包含可能造成运行中断的所有重要背景信息（例如，服务高峰时段，大负载）。最后，在未来避免事故的建议应该包含列表中每个建议所提交的单据或者缺陷 ID。

附录 D 的 D.3 节中可以找到一个事后剖析模板示例。如果组织没有事后剖析模板，可以使用这个作为基础。

摘要应该包含最基本的信息——事故何时发生，根源是什么，应该重申需要预算批准的任何建议，以便高管人员在他们心中，将预算申请和事故关联起来。

因果分析（Causal Analysis）或者促成条件分析（Contributing Conditions Analysis）找出引起停机的条件。这有时候被称作根源分析（Root Cause analysis），但是该名称暗示运行中断只有一个根源。如果传统或者办公室政治需要使用“根源”一词，至少要使用复数（causes），以强调有许多可能的根源。

虽然指向单一根源可以满足情感上的需求，但是现实是造成停机的因素很多。运行中断有单一根源的信念会产生这样一种暗示：运营是一系列多米诺骨牌，推翻一块就会造成停机。现实比这复杂得多。正如 Allspaw（2012a）的文章《Each Necessary, But Only Jointly Sufficient》（每个因素都是必要的，但是只有合在一起才是足够的）所指出的，找出故障的根源就像找出成功的根源。

事后剖析的传达

一旦完成了事后剖析报告，应该将拷贝发送到对应的团队，包括参与修复运行中断的团队和受到影响的人员。如果用户在公司外部，应该制作一个删除专有信息的版本。小心不要违反公司关于外部沟通的政策。外部版本可以做明显的简化，向外发布事后剖析可以建立客户的信心，是一种最佳实践。

外部沟通时，事后剖析报告应该搭配技术性较低的简介，强调重要的细节。大部分外部客户不理解技术性的事后剖析。包含具体的细节，如开始和结束时间、影响的人或事、出现哪些问题以及吸取的教训。阐述你将在未来利用这些经验改进工作。如果可能，加入人为因素，如巨大的努力、不幸的巧合以及有效的团队工作。还可以加入从这一经历中学到的其他知识。

沟通的真实性很重要，承认故障，要用人类的语言描述，而不是采用新闻机构的腔调。图 14.1 是好的外部沟通的一个例子。注意，报告用第一人称撰写，带有真正的懊悔之情——不做任何隐藏。避免使用第三人称隐瞒真相，或者用“我们对事故可能给用户和客户带来的影响表示遗憾”等说法淡化运行中断的所有影响。不要对“可能”带来的影响表示遗憾，而是确实发生了影响——否则你就不会发送这一消息了。在 Transparent Uptime 博

客 (Rachitsky 2010) 上的帖子 “A Guideline for Postmortem Communication” (事后剖析沟通方针) 可以找到更多建议。

主题：CloudCo 2014 年 2 月 4 日事故报告：DNS 服务短暂中断

上周二 (2 月 4 日), CloudCo 经历了一次运行中断, 我们的 DNS 服务中断了将近 4 分钟。由于这次故障, 在恢复整个服务期间, 客户遭遇了 4 分钟的停机时间。我要因为停机对客户日常操作的影响表示歉意。对我们来说, 未事先计划的任意长度停机时间都是不可接受的。在这个案例中, 我们没能达到客户和自己的期望, 为此, 我真的感到内疚。

我将花一小段时间解释停机的原因、停机期间发生的情况以及我们为避免未来发生相同情况所作的努力。

技术摘要

CloudCo 经历了 4 分钟的 DNS 运行中断。这个问题是 DNS 速率限制工具中意外输入的结果, 这一输入造成设置 0 查询 / 秒的速率限制。网络运营团队立刻发现了这个问题, 在 2 分钟内确定速率限制, 并在下一分钟恢复配置。在事故开始后的 4 分钟内, 全网范围内的 DNS 服务恢复正常。

事故描述

太平洋标准时 17: 10 (01:10 UTC), CloudCo DNS 服务无法响应 DNS 查询。在应用速率限制的 4 分钟内, 对 CloudCo DNS 的所有 DNS 查询都被丢弃。这影响了所有发送到 BIND 的 DNS 查询。

事件时间轴——2014 年 2 月 4 日

17:10 PST (01:10 UTC) – CloudCo DNS 服务应用了一个错误的速率限制

17:11 PST – 内部自动警报

17:13 PST – 发现 DNS 速率限制问题

17:14 PST – 删除错误的 DNS 速率限制

17:14 PST – CloudCo DNS 确认恢复正常状态

17:17 PST – CloudCoStatus 推文状态更新

解决方案

在 4 分钟内发现错误的更改并更正。

建议

- ☐ 修订应用全局 DNS 速率限制的内部过程, 加入同行审核。
- ☐ 实施附加软件控制, 校验工具的用户输入。

图 14.1 运行中断之后发送给客户的事后剖析邮件示例

14.4 警报的定期审核

警报日志应该定期审核, 以发现趋势, 分配资源进行长期修复, 最终减少接收警报的总数。实施这种策略时, 警报不仅是认识问题的途径——它们会变成改进系统稳定性的主要手段之一。

应该有一种系统性的方法减少警报数量, 否则无序状态可能导致警报随着时间推移而变得越来越频繁, 直至失去控制。还应该分析警报的趋势, 因为每个警报都是更大问题的潜在信号。如果我们的目标是持续改进系统, 就必须抓住一切机会寻找需要改进目标的线索。

举行每周会议审核警报和问题并搜索趋势, 是很有益的。在这种会议上, 可以找出解决较常见问题的项目, 了解生产环境的整体健康状况。季度审核对发现更大的趋势很有用,

可以纳入季度项目规划周期。

警报日志应该由接收者加以注释。大部分系统可以用关键词标记警报，然后分析关键词找出趋势。下面列出一些关键词的例子：

```
cause:network
cause:human
cause:bug
cause:hardware
cause:knownissue
severity:small
severity:medium
severity:large
bug:BUGID
tick:TICKETNUMBER
machine:HOSTNAME
```

这些关键词使你可以注释警报的总体根源、严重性和相关的缺陷及单据 ID。可以使用多个标签。例如，由硬件故障导致的小规模运行中断可以标记为 `cause:hardware` 和 `severity:small`。如果问题是由已知缺陷造成的，警报可能标记为 `cause:knownissue` 和 `bug:12345`（假定已知问题的缺陷 ID 为 12345）。对于管理层已经决定不修复某个缺陷或者缺陷修复正在进行中、在修复交付之前采用变通手段的情况，这样标记的项目将被保留。

作为分析的一部分，应该生成一个报告，说明最常见的警报根源。寻找具有相同缺陷 ID 或者单据号码的多个警报。还要搜索最严重的运行中断并进行特殊审查，检查事后剖析报告和建议，看看根源或者修复能否聚合或应用到超过一个运行中断。

14.5 收到太多传呼

少量警报是合理的，但是如果警报的数量变得太大，可能需要干预。对于不同团队，造成过多警报的原因不同。应该共同协商、制定一个容限。

如果容限不断被超越，情况变得越来越糟糕，可以考虑如下的干预措施：

- ❑ 如果已知的缺陷在一段时间（比如两个发行周期）内造成频繁的传呼信号，未来这个警报应该自动导向开发人员的班次。如果没有开发人员班次，则推进这种班次的建立。这能够统一开发人员和运营人员的动机，促进问题的解决。
- ❑ 如果通过传呼机接收到的警报和 SLA 的维护不直接相关，应该将其从生成传呼信号的警报更改为在故障单据系统中生成单据。
- ❑ 就特定问题会见开发人员。确保他们理解问题的严重性。建立修复最频繁或者反复出现问题的共同目标。如果这是文化的一部分，建立缺陷列表，召开缺陷扫除大会或者修复周。
- ❑ 协商临时降低 SLA。相应调整警报。如果警报容限已经和 SLA 不一致（也就是接收到低优先级问题的警报），则重新保持一致。协商临时调降终止的条件，这可能是

固定时间（如一个月）或者可计量的条件（如当后续的3个发行版本投放没有发生故障或者回滚时）。

- 如果上述手段都失败，则发出黄色预警：允许团队推迟所有其他工作，直到情况改善。建立成功的计量指标，并努力达成该目标。

14.6 小结

作为服务维护任务的一部分，我们必须有处理异常情况的手段。为了确保问题得到正确的处理，建立轮值制度。

值班人员应该包括运营人员和开发人员，以统一运营优先级。设计值班时间表有许多方法：每周、每天或者每天多个班次。目标应该是在每个班次中不产生太多警报，使后续工作能够完成。

通知值班人员的方式很多。通常，警报的共享通过向手持设备（如电话加上其他冗余机制）发送消息实现。

在值班之前，应该完成值班准备检查列表。值班人员应该可以联络到，以防警报的出现。否则，他们应该正常工作和休息。

一旦接收警报，值班人员的头等大事是解决问题，即使这意味着实施快速修复，而将长期修复留待之后进行。

值班剧本记录响应各种警报的措施。如果该文档不足，问题应该升级到服务负责人或者其他升级轮次。

警报应该记入日志。对于重大警报，应该撰写事后剖析报告，记录发生的情况、为解决问题所采取的措施，以及为未来避免问题可以采取哪些措施。

警报日志和事后剖析应该定期审核，确定趋势并选择解决系统性问题、减少警报数量的项目。

练习

1. 值班系统的主要设计元素是什么？
2. 描述你的当前值班策略，你是不是值班轮次的一部分？
3. 值班人员在接收警报时如何改变优先级？
4. 组织中值班任务的先决条件是什么？
5. 指出你所监控的10项活动。对于每项活动，适合采用哪些类型的通知，为什么？
6. 事后剖析包含哪四种元素，每种元素需要哪些细节？
7. 为你所涉及的一次事故撰写事后剖析报告。
8. 值班系统是如何逐步改善运营状况的？



第 15 章 Chapter 15

灾难准备

跌倒不算失败，拒绝振作才是失败。

——西奥多·罗斯福

灾难和重大运行中断都可能发生，公司上下每个人都需要承认这一事实，培养接受运行中断并从中吸取教训的心态。运营组织必须很好地处理运行中断，避免重复过去的错误。

前面我们已经研究了与故障和运行中断弹性相关的技术，以及值班等组织策略。本章我们将讨论个人、团队、规程和组织层面上的灾难准备。人们必须经过培训，对规程有足够的了解，充满信心地执行规程。团队必须一起实践，建立团队的凝聚力和信心，找出并解决规程上的问题。组织必须进行实践，找出团队间的缺口，确保整个组织都做好了处理意外情况的准备。

每个组织都应该有在所有层面上确保灾难准备的策略。在个人层面，应该通过正式手段（书籍、文档、辅导）和游戏进行培训。团队和组织应该利用应急演练和游戏日演练改善过程，找出覆盖范围中的缺口。应该使用后面描述的事故指挥系统（Incident Command System, ICS）等模型协调运行中断的恢复工作。

成功运营大型分布式系统的公司（如 Google、Facebook、Etsy、Netflix 等）意识到处理运行中断的正确方式是做好准备，采用在未来减少运行中断的做法，并通过实施有效的响应规程降低风险。在《Built to Win: Deep Inside Obama's Campaign Tech》（为胜利而战：深入奥巴马竞选活动中的技术，Gallagher, 2012）中，我们可以学习到，即使在总统竞选中，游戏日演练也是成功的关键。

15.1 心态

灾难准备的第一步是承认灾难和重大运行中断可能发生。它们是业务的正常和意料之中的部分。因此，我们要为此做好准备，并在它们发生时做出正确的响应。

我们希望减少运行中断的数量，但是完全消除它们是不现实的。任何技术都不完美。没有一种计算机系统不会发生故障。消除最后 0.000 01% 的停机时间比消除前面 99.999 9% 要昂贵得多。因此，作为一种权衡，我们容忍一定数量的停机时间，并用处理故障情况的准备加以平衡。

同样，人无完人。每个人都会犯错误，我们致力于少犯错误，并且绝不两次犯同样的错误。我们试图招聘既一丝不苟又有创造力的员工，开发过程和规程试图在错误造成运行中断之前捕捉到它们，并且在事故发生时尽可能好地处理。正如 14.3.2 节中所讨论的，每次运行中断都应该视为从我们自己和其他人的错误中学习并改进系统的机会。运行中断暴露了落点，是我们找到使系统更具弹性、添加预防性检查的地方，教育整个团队以便他们不再犯相同的错误。这样，我们将建立起能够对抗脆弱的组织和服务。

发生重大事故时找出某人加以指责并将其解雇是达不到预期目的的，但是在某些公司是常见的做法。当人们害怕被解雇时，他们会采取和出色运营相反的做法。他们将掩盖错误，降低透明度。当真正的根源模糊不清，没有人能从错误中吸取教训，无法进行进一步的检查，这样问题就可能再次发生。这就是 DevOps 文化的一部分是接受故障并从中学习的原因，这样可以暴露问题，从而解决问题。

遗憾的是，我们常常看到当大型公司或者政府网站出现广为人知的运行中断时，管理层匆忙地解雇某人，以向所有人表明事件得到了认真的处理。有时候，媒体会激化这种情况，要求某人承担责任和解职，对尚未做出这一举动提出质疑。媒体可能最终会指责 CEO 或者总裁不开除某人。最好的方法是发布公开版本的事后剖析报告（在 14.3.2 节中讨论），不指责个人，而是将重点放在所吸取的教训以及为避免再次发生所采取的附加检查措施。

15.1.1 反脆弱系统

我们希望分布式计算系统是反脆弱（Antifragile）的。反脆弱系统在压力越大或者暴露在随机行为下时更加强健。弹性系统可以在压力和故障下存活，但是只有反脆弱系统在遇到逆境时会变得更强大。

反脆弱不是脆弱的反面。脆弱的物体当暴露在压力之下时会破坏或者变化。因此，脆弱的反面是面对压力保持不变的能力。茶杯是脆弱的，如果不小心轻放就会打破。反面则是跌落后保持不变（不会破碎）的茶杯。相比之下，反脆弱的物体通过变得更强健对抗压力。例如，炼钢的过程使用可以摧毁大部分物体的高温，但是这一过程却使钢变得更具强度。神话中的九头蛇是反脆弱的，因为每当失去一个头，它就会长出两个。我们的学习方法是反脆弱的：为考试而学习会使大脑辛勤工作，从而强化了它的能力。

脆弱系统在意外发生时会被破坏。因此，如果我们想要建立反脆弱的分布式计算系统，应该引入随机性，频繁而主动地触发弹性功能。每次故障都导致团队将数据库故障切换到一个副本，团队在执行这一规程时就更加熟练。他们还会因此得到改进的思路。更好的执行和规程改进使系统更加强大。

为了加速这类改进，我们人工引入故障，不是尝试避免故障而是挑动故障。如果故障切换过程失败，我们应该以受控的方式研究这一事实，最好是在尽可能多的人清醒且可以响应的正常工作时间内进行。如果没有以受控的方式触发故障，就只有在真正的紧急情况下才能研究这些问题：那通常是在下班时间，大家都在睡觉。

15.1.2 降低风险

以受控方式实践高风险规程，使我们能够降低风险。不要将高风险行为和高风险规程混为一谈。高风险行为从定义上是危险的，应该避免。高风险规程可以改进。例如，粗心大意是应该避免的高风险行为。除了停止粗心大意之外，没有其他方法能降低风险，这种风险是固有的。但是，Web 服务故障切换的规程可能有很大的风险，也可能没有什么风险，如果它有很高风险，可以加以改进和重新设计，降低风险。

如果我们将高风险行为和高风险规程混为一谈，最终会将好的做法应用到错误的事物上。我们避开应该重复执行直到消除风险的规程。通过重复改进事务的技术术语是“实践”——熟能生巧。

传统的思想是：计算系统是脆弱的，必须加以保护。结果是，不管是管理策略还是技术功能，都包围在保护系统之中。我们应该投入反脆弱实践，提高系统及组织的强度和信心，而不是在运营系统之外建起壕沟。

对故障的这种新态度保持了管理实践和复杂系统事实的一致性：

- ❑ 新软件发行版本总是有新的意外缺陷。
- ❑ 识别根源的意图不是追究责任，而是了解改进系统和运营实践的方法。
- ❑ 在不可靠的组件基础上构建可靠的软件意味着弹性功能是必需的，而不是无意义的负担、“锦上添花”的功能或者浪费。
- ❑ 在云规模上，复杂的故障是不可避免、无法预测的。
- ❑ 在生产环境中，复杂系统往往以无法事先明确知晓的方式交互（超时、资源争用、移交）。

理想情况下，我们喜欢永远正常运行的完美系统。遗憾的是，这种系统只存在于销售宣传资料上。在这种系统出现之前，我们宁可用足够的故障确保对所采取预防措施的信心。不管是自动还是手动的，故障切换机制都需要演练。如果是自动的，该机制未被激活的时间越长，我们对其正常工作的信心就越弱。系统可能已经改变，可能意外地和故障切换机制不兼容或者破坏该机制。如果故障切换是人工过程，我们不仅会对规程失去信心，还会对团队实施规程失去信心。换言之，团队缺乏实践，或者其知识集中在少数几个事项上。

理想的情况下，我们希望服务故障足以维护对故障切换规程的信心，但是又不会危害服务本身。因此，如果某个组件过于完美，最好是人为产生故障，重新确立信心。

案例研究：重复高风险行为以降低风险

在一家公司里，每个人都知道最后一次数据库需要故障切换时进行得并不顺利。因此，团队害怕实施这个规程并且避开它，认为这样才是好的风险管理。这实际上增加了风险。如果紧急情况下需要这一过程，就不太可能顺利实施。意识到这一事实之后，经理要求团队每一周都使服务失效一次。

前几次很艰苦，但是整个团队都在运营人员执行数据库故障切换时观察和提出意见。规程手册按照所提意见进行更新，记录预先检查手段。一位团队成员指出，在故障切换之前，她验证磁盘有足够的空闲空间，因为以前出现过相关的问题。团队其余成员不知道这个预先检查手段，但是现在该手段已经被添加到规程手册，每个人都知道需要这么做。

更重要的是，这一发现提出了一个重要的问题：为什么没有始终监控空闲磁盘空间？如果需要紧急故障切换时磁盘空间太少会发生什么情况？这催生了一个编外项目——监控系统可用磁盘空间。许多类似的问题被发现并解决。

最终，这一过程变得更可靠，很快增加了人们的信心。团队消除了一个压力源。

15.2 个人培训：灾祸之轮

系统发生破坏有许多方式。值班人员应该处理大部分常见破坏，但是需要对自身的能力有信心。我们通过提供许多不同方式的培训来建立信心：文档、辅导、跟随更有经验的人以及实践。

灾祸之轮（Wheel of Misfortune）是运营团队为值班人员做准备的一个游戏。这是改善个人处理值班任务知识和共享最佳实践的手段。这个游戏还可以用于向整个团队介绍新规程，帮助团队成员保持技能，学习必要的新技能以及相互学习。游戏的玩法如下。

整个团队在一间会议室里开会。每一轮由一位志愿者作为选手，另一位志愿者作为灾星（Master of Disaster, MoD）。MoD 解释值班中的一个情况，选手解释如何解决问题。MoD 作为系统，响应选手所采取的任何措施。MoD 只在足够的猜测或者选手走进死胡同之后才给出解决问题的提示。最终，测试者解决问题，或者由 MoD 解释问题应该如何解决。

MoD 通常从说出一个接收来自监控系统的警报及识别消息开始。选手可以说出应该检查的仪表盘或者日志作为回答。MoD 解释在仪表盘或者日志上看到的情况。这种一问一答的形式持续下去，选手首先确定问题所在，然后经历解决问题的各个步骤。如果 MoD 准备了屏幕截图（真实或者虚构的），在选手提出合适的问题时展示，对整个游戏大有帮助。这些截图可能来自真实的事故或者从头开始准备，无论哪一种都可以给演练带来真实性。

有些团队在“灾祸之轮”游戏中采用“无便携电脑”规则。选手不允许使用便携电脑直接检查仪表盘，而是必须对一位队友说出想查看或者做的事情，由后者使用便携电脑，将结果投影到其他人都能看到的共享屏幕上。这迫使选手明确知道自己所要做的操作以及原因，向其余观众揭示选手的思维过程。

如果选手是个新人，应该使用基本场景：最常见的警报和最典型的解决方案。较为高级的选手应该采用设计更细微问题的场景，以便观众学习。游戏的目标绝不是愚弄选手，或者提供必败的结局（也称为“小林丸”情景）。MoD 的目标是培训工作人员，而不是为了击败对手“赢”得游戏。

为了保持游戏的趣味性，团队可以增加游戏的舞台效果，在开始时演奏主题音乐或者用一个转盘选择下一个场景、制造效果。

灾祸之轮是纯粹的培训活动，所以不进行计分。团队领导人或者经理应该观摩游戏，作为辨别哪些成员应该进一步培训的手段。没有必要为难团队成员，进一步培训的建议应该私下做出。值得肯定的是，这个游戏也是识别哪些团队成员具有很强故障维修技能的手段，这些成员可以多加培养，以帮助其他人开拓自身能力。

灾祸之轮游戏应该在团队有新成员或者新技术加入时更频繁地进行，而在一切保持不变时降低频率。将培训过程转变成游戏，人们就更有动力参加。这种游戏使培训过程变得有趣，吸引整个团队的参与。

15.3 团队培训：应急演练

应急演练（Fire Drills）练习特定的灾难准备过程，在这些情况下，触发真实故障，主动测试涉及的技术和人。

建立弹性系统的关键是接受故障的发生，并投入准备以更快、更有效地响应这些故障。未经测试的灾难恢复计划完全不是真正的计划。应急演练是抢先触发故障，观察、修复故障，然后重复，直到过程完善、涉及人员对自己的技能建立信心的过程。

演练的效果在于让我们实践并找出规程中的缺陷。在密切注视下于生产环境中导致故障，比希望在不监控的情况下系统正确表现的策略更好。在生产环境中进行这些演练确实有发生灾难的危险。但是，灾难发生在整个工程团队都做好响应准备的情况下不是更好吗？

因为找出了缺陷并进行修复，演练能够建立对灾难恢复技术的信心。故障切换机制越不经常使用，我们的信心就越弱。想象一下某个故障切换机制一年以上都没有触发过的情況。我们不知道环境中是否有不相关的变化导致该过程过时。期待它无缝工作是不合理的，无知可不是福音。不确定故障切换是否有效会带来压力。

演练在运营团队中建立信心。如果团队不习惯于处理灾难，他们就更有可能会过快反应、以拙劣的方式反应或者感觉到不应有的压力和焦虑。这降低了很好地处理灾难情况的能力。

演练可以为团队提供机会，冷静、自信地应对。

演练可以建立管理层对运营团队的信心。虽然有些高管人员宁愿保持无知，但是聪明的高管知道故障确实会发生，公司做好准备并进行演习是最好的防御手段。

演练可以增加对单独过程、涉及重要系统的较大测试甚至涉及整个公司的更大型多日测试的信心。从小规模的演练开始，逐步升级到最大规模的演练。如果你不通过一系列较小、不断提升的演练积累能力和自信，尝试大规模演练是无效的。

15.3.1 服务测试

选择单独的故障切换机制进行测试，是非常实用的起点。找出不常演练的故障切换机制并选择其中一个。这些测试应该是封闭的，如果发现问题，只需要直接的团队成员响应。通常，这些测试会发现不完整的过程、文档的不足、知识的缺失和配置问题。

演练包括导致触发故障切换机制的故障。这可能包含关闭某台机器电源（拔出电源线）、关闭虚拟机、杀死运行中的软件、断开网络连接或者其他可能中断服务的手段。

策略之一是使用演练来改进过程。如果过程不可靠或者未经测试，让同一个人反复进行演练，在每次迭代中改进过程文档和软件是值得尝试的。通过让同一个人进行每次迭代，他们将很好地了解系统，做出大规模的改进。他们可能在一天中进行许多次迭代，寻求微调过程的方法。另外，更进一步进行迭代可以修复更深层次的问题。

另一种策略是利用演练增强团队成员个人的信心。每次迭代可以由不同人进行，直到每个人都成功地完成过程（可能不止一次）。

混合方法由同一个人进行演练直到过程稳定，然后采用单独团队成员的策略，验证团队中的每个人能够完成该过程。一旦实现了这一水平的改进，演练的频率就可以低一些。可以选择在一段时间内过程没有被真正的故障触发、过程更改或者新成员加入团队时进行演练。

选择在哪一天进行演练需要认真的思考。最好是在没有重大销售展示活动或者新产品发行计划的日子进行。演练时应该每个人都有空闲，最好不要选择有人刚休假归来或者将要休假的日子。

Google Chubby 停机演练

Google 内部有一个称作“Chubby”（“小胖子”）的全局加锁服务。由于可靠性很好，它具有很高的声誉，导致其他团队错误地认为它是完美的。一次小规模的运行中断给编写代码时假定 Chubby 不会失效的团队带来了很大的问题。

Chubby 团队不希望鼓励错误的编码方法，决定最好是由公司有意造成停机。如果一个月中没有至少几分钟的停机时间，他们就有意地停止 Chubby 五分钟。这种停机安排预先做了充分的通知。

第一次计划内的停机在即将开始之前被取消了。许多关键项目报告它们无法承受这

种测试。

这些项目团队得到 30 天的时间修复自己的代码，但是公司警告不能有进一步的延迟。现在，Chubby 团队得到了重视。此后一直按照规划进行停机演练。

15.3.2 随机测试

另一种策略是测试各种不同的潜在故障。这种策略不选择改进特定的故障切换过程，而是随机选择机器或者其他故障域，引起故障，验证该系统是否仍然运行。这可以在计划好的一天或者一周中进行，或者连续进行。

例如，Netflix 创建了许多自治代理，每个都经过编程以造成不同类型的运行中断。这些代理被称作“猴子”，一起组成 Netflix 猴子军团（Simian Army）。

“混沌猴子”（Chaos Monkey）随机中断虚拟机的运行。它被编程为以不同的概率选择不同机器，有些机器完全被排除。每小时，“混沌猴子”苏醒，随机选择一台机器，将其中止。

“混沌大猩猩”（Chaos Gorilla）选择整个数据中心，模拟网络分段故障或者整体故障。这会导致严重的破坏，恢复需要复杂的控制系统以重新平衡负载。因此，这个代理作为计划测试的一部分人工运行。

猴子军团不断壮大。新成员包括“延迟猴子”（Latency Monkey），它招致人为的 API 调用延迟，模拟服务降级的情况。猴子军团的详细描述可以在《The Antifragile Organization》（反脆弱组织）中找到（Tseitlin 2013）。

这种演练开始时有较大风险，因为它们跨越了许多个故障域。因此，应该得到管理层的批准。说服管理层理解这类测试的价值可能很困难。大部分经理都希望回避问题，而不是招致问题。重要的是从改进问题（这些问题是不可避免的）响应能力的角度出发，强调找出问题的最佳时机是在受控的环境中，而不是员工们入睡的深夜。将过程与涉及整体服务正常运行时间的业务目标联系起来，这样做还会提高团队成员的士气和信心。

15.4 组织培训：游戏日 /DiRT

游戏日演练是在多日中进行的组织范围灾难准备测试，涉及许多团队，往往包含沟通、后勤和财务等非技术部门。游戏日演练的重点是测试复杂的场景，尝试系统和团队间很少测试的接口，识别未知的组织依赖性。

游戏日演练可能涉及数据中心的多日运行中断、一周内的复杂网络及系统停机，或者在主要团队于一段时间内消失的情况下验证辅助人员能否成功运营服务。这些演练还可以用于准备即将到来的事件，在这些事件中预计会有额外的工作负荷，团队处理大规模运行中断的能力至关重要。例如，在 2012 年选举日之前的数周内，奥巴马竞选团队举行了 3 次

全天会议，测试“拉票”行动。这一演练在《When the Nerds Go Marching in》(当书呆子们大游行时，Madrigal 2012)中有详细的描述。

因为规模和范围较大，此类测试可能更紧凑，防止更大的运行中断。当然，因为规模和范围更大，此类测试也需要更多的规划、更多的基础设施和更高级别的管理层批准及支持。

组织必须相信通过学习得到的价值超过演练的成本。游戏日演练可能是一项大规模的技术工作，涉及数百个人·天的工作量。演练中还有可能造成真正的运行中断，蒙受经济损失。

高管必须承认所有系统都不可避免地会发生故障，信心最好通过实践（而不是回避）获得。他们应该理解，最好是在受控环境下、关键人员清醒时发生故障。学习故障切换系统的时机不应该是凌晨四点，此时关键人员正在睡觉或者休假，这是应该避免的风险。

Google 的灾难恢复测试（Disaster Recovery Testing, DiRT）是该公司的游戏日演练形式。DiRT 的规模很大，重点是测试团队之间的互动。这些互动通常不太频繁演练，但是在较大的灾难时很有必要。这些演练比基于团队的服务测试和随机测试更可能导致客户可见的运行中断和收入损失。因此，团队应该在很好地完成自己的特定应急演练之后才能参与 DiRT。

15.4.1 开始

DiRT 演练可能规模很大，但是最好是从小规模开始，向整个公司介绍其概念。小项目更容易向管理层证明。可正常工作的小型示例比假设性的大项目更容易得到批准，老实说，大项目对不熟悉概念的人来说很可怕。当应该保证系统正常运行的人告诉高管人员希望将很大一部分系统停机时，高管人员会感到非常担心。

从小项目开始还意味着测试更简单。Google 的 DiRT 开始时只涉及几个团队。测试很安全且经过设计，不造成任何用户可见的中断。即使意味着测试不是很有用也要这么做，因为这样可以使各个团队习惯于 DiRT 的概念，确保他们吸取的教训能够用于对系统的建设性改进，并且让他们知道故障不会变成相互指责或者搜寻罪人的过程，还使 DiRT 测试协调人得以保持过程简单，尝试其方法论和跟踪系统。

Google 最早的网站可靠性工程（SRE）团队位于加利福尼亚州山景城的公司总部。最终，Google 在爱尔兰都柏林增加了 SRE 团队，处理夜间值班任务。第一次 DiRT 演练简单地测试都柏林的 SRE 能否在一整周内独立运营服务。山景城的团队伪装成不可用，这暴露出都柏林 SRE 依赖一个事实：如果他们碰到问题，最终山景城的 SRE 会被叫醒，协助解决。

Google 早期的另一个简单测试是尝试在无法访问源代码库的情况下工作一天。这一演练发现某些领域中，生产过程所依赖的系统并没有按照始终可用的目标设计。实际上，许多测试都涉及验证生产系统所依赖的系统全部构建于生产系统关键路径上。组织越大，这

些依赖性越有可能只能通过主动测试发现。

15.4.2 扩大范围

随着时间的推移，测试可能扩大到包含多个团队。可以提高测试目标的门槛，包括风险更大的测试、实时测试和移除低价值的测试。

今天，Google 的 DiRT 过程可能是世界上此类演练中规模最大的。到 2012 年，涉及的团队数量增长了 20 倍，覆盖所有 SRE 团队和几乎所有服务。

将过程增长到这种规模依赖于一种文化，在这种文化中，问题的识别被视为理解系统如何改进的正面手段，而不是恐慌与相互指责。有些运营团队除了服务交付平台的持续交付系统已经提供的功能之外，看不到测试的其他好处。预测团队开始有参与意愿的最佳指标是：之前的故障是造成对根源的搜索和修复，还是造成对某人的指责？指出较早、较小的成功，可以使新团队和高管有信心扩展项目。

复杂测试的例子可能包括模拟地震或者其他造成公司总部不可用的灾难。可以禁止公司总部的任何人与公司其余分部通信。Google DiRT 曾经进行过这一测试，发现远程站点可以继续工作，但是紧急采购（如备份发电机的燃料）的批准链需要公司总部人员参与。这些关键的发现不是技术性的。另一个非技术性发现是，如果所有测试让总部人员无事可做，他们将涌入自助餐厅，造成餐厅的 DoS 洪泛。

企业应急通信计划也应该进行测试。在大部分运行中断中，人们可以使用常规的聊天室等手段通信。但是，在整个网络失效时需要某种紧急的通信机制。第一次 Google DiRT 演练发现，只有一个人能找到紧急通信计划，出现在正确的电话会议中。现在，定期的应急演练抽样检查每个人是否有正确的信息。在后续的演练中，超过 100 人可以找到并执行紧急通信计划。此时，Google 发现电话会议只能支持 40 个呼叫者。在另一次演练中，一个呼叫者在电话会议中使用了保持通话功能，使“等待音乐”淹没整个电话会议，令其无法使用。这识别出了将某人“踢”出会议的需求。所有这些问题都是在模拟灾难中发现的。如果在真实的紧急情况下发现，那就是真正的灾难了。

15.4.3 实施和后勤

有两种类型的测试：全局测试涉及重大事件，如由策划团队发起的数据中心停运；团队测试由单独团队发起。

事件可能持续数天，或者仅发生在“糟糕”的一天里。Google 安排一整周的时间，但是 DiRT 活动在测试目标满足之后立刻结束。为了保证每个人做好准备，DiRT 不在公告的开始时间启动，而是有一个小的延迟，延迟的长度保密。

大的事件有许多可变因素需要协调。协调人应该兼具技术和项目管理经验，可以在项目中投入足够的时间。非常大规模的协调和规划可能需要一个全职人员，尽管并不是 12 个月中都有事件发生。一年的大部分时间都将花在测试的规划和协调上。剩余的几个月则花

在结果的审核和组织范围改进的跟踪上。

规划

事件规划提前几个月就开始了。团队需要时间决定应该测试的项目、选择监督人并构造测试方案。监督人负责设计和执行测试。在关键日期之前很久，他们就在文档中记录测试目标、场景和遵循的脚本，设计测试。例如，脚本可能包括致电某个服务的值班人员，由他模拟某种情况，就像灾祸之轮演练一样。公司也可能计划主动停止某个系统或者服务，观察团队的反应。在实际测试中，监督人负责测试的执行。

尽早知道测试活动日期，团队就可以安排项目工作和假期。团队可能还要利用这些事件进行单独的演练，使测试活动能够集中在寻找团队间缺口上。如果团队自己的过程还没有很好地实践，DiRT 也不会顺利。

在 Amazon 的第一个“游戏日”之前，John Allspaw 进行了一系列公司范围的推介，让每个人都了解即将进行的测试。他指出测试活动达到摧毁一整个数据中心的规模。人们不知道将被摧毁的是哪一个数据中心，使得准备工作更加全面（Robbins、Krishnan、Allspaw & Limoncelli 2012）。

所有测试计划都预先提交给跨职能专家团队审核和批准，以缓解风险。这个团队检查不合理的风险。以前从未进行过的测试风险更大，应该在沙箱环境中或者通过模拟进行。某些系统准备不足，在运行中断中无法存活，这种情况往往预先已经得知。首先使这些系统失效，将它们标记为无法通过测试，在活动中不涉及它们。加入它们不能学习到任何经验教训，这些机器应该列入白名单，使它们在活动中仍然能够接受服务。例如，如果用网络过滤模拟运行中断，这些机器可以排除在过滤器之外。

组织

测试活动的成功需要两个子团队。技术团队设计全局测试并评估团队测试计划。测试按照质量、影响和风险分级。在测试活动期间，这个团队负责引发全局运行中断、监控以及确保情况不会失控。该团队还要处理测试导致的不可预见问题。

协调团队负责规划、时间安排、预算和执行。它和技术团队一起避免测试相互冲突。协调团队的成员验证所有准备工作完成。他们还负责与管理层和整个企业范围内的沟通。

在测试活动中，两个子团队都驻扎在指挥中心，指挥中心的作用是某种“任务控制”。由于测试的特性，指挥中心必须是一个物理位置，因为依赖虚拟会议空间过于危险。对于大型组织，这涉及相当多的差旅。

指挥中心人员致电监督人，告诉他们开始测试。测试控制人处理意外的问题，并向全体人员传达测试活动的状态。他们监控测试的进度，以及测试对不应该受影响系统产生的影响。测试控制人在出现较大问题时可以暂停所有测试。

指挥中心内最有价值的角色之一是“讲故事的人”。这个人编造和叙述灾难。他应该

创作一个故事，故事中不切实际的成分足以让人们知道这是一次测试。例如，故事可能涉及僵尸攻击，邪恶的天才对总部的所有人实施催眠，或者具有神秘力量的一个错误预言家。故事的推进应该每天通过电子邮件或者老式无线电“广播”发布更新。最后一次公告应该结束故事，让每个人知道测试完成，僵尸已经被打退，或者其他情况。

讲故事的人有助于化解测试的辛苦。如果这个人干得出色，人们可能会盼望下一年的测试活动。

建议阅读

许多年来，DiRT 和相关的方法是公司的秘密。但是，2012 年一些公司和从业人员在一系列文章中解释了他们的灾难测试规程，Tom 提炼了这些文章的精华，发表在《ACM Queue》杂志上：

- ❑ “Resilience Engineering: Learning to Embrace Failure”（弹性工程：学会接受故障）是 Tom 发起的一次小组面谈，参与者有 Google DiRT 项目协调人 Kripa Krishnan、Amazon 游戏日架构师 Jesse Robbins 以及 Etsy 技术运营高级副总裁 John Allspaw（Robbins 等人，2012）。
- ❑ Krishnan 在 “Weathering the Unexpected”（承受意外情况）中详细介绍了 Google 的 DiRT 项目（Krishnan，2012）。
- ❑ Allspaw 在 “Fault Injection in Production”（生产环境中的故障注入）解释了 Etsy 项目的理论与实践（Allspaw，2012）。
- ❑ Tseitlin 在 “The Antifragile Organization”（反脆弱组织）中详细介绍了 Netflix 猴子军团，解释它如何增强弹性，最大化可用性（Tseitlin，2013）。

后来，Steven Levy 获准亲自观摩 Google 年度 DiRT 过程，为《Wired》杂志撰写了一篇题为《Google Throws Open Doors to Its Top-Secret Data Center》（Google 打开了绝密数据中心的大门，Levy，2012）的文章。

在 2012 年美国 总统 竞选 之后，《The Atlantic》杂志的文章《When the Nerds Go Marching in》描述了奥巴马竞选团队在准备 2012 年选举日而进行的游戏日演练（Madrigal，2012）。Dylan Richard 的谈话 “Gamedays on the Obama Campaign”（奥巴马竞选团队的游戏日）提供了第一手报道，说明科技不一定能赢得选举，但是确实可能使其失败（Richard，2013）。

15.4.4 经历 DiRT 测试

下面是从参与测试的工程师视角虚构化叙述的 Google DiRT 演练。其中的姓名、地点和情况已做了更改。这一描述改编自 Tom 为《ACM Queue》杂志撰写的文章题为《Google DiRT: The View from Someone Being Tested》（Google DiRT：被测者的观点）（Limoncelli，2012）。

[电话响起]

Tom: 你好?

Rosanne: 你好, Tom, 我正在监督一项 DiRT 演练, 你是 [服务名称] 的当值人员吗?

Tom: 是的。

Rosanne: 在这次演练中, 我们假定 [服务名称] 数据库需要从备份中恢复。

Tom: 好的, 这是真正的演练吗?

Roanne: 不, 只需要告诉我过程。

Tom: 好, 我将按照运营文档的指导进行。

Roanne: 你能找到文档吗?

[Tom 在键盘上按了几下]

Tom: 是的, 我找到了。

Rosanne: 好, 启动服务的一个克隆, 将数据库恢复到它上面。

在接下来的几分钟, 我有两个发现。首先, 文档中的一个命令现在需要附加参数。其次, 进行恢复的临时区域没有足够空间。在编写规程时它有足够的空间, 但是数据库在之后扩大了。

Rosanne 提交了一个缺陷报告, 请求更新文档。她还提交了一个缺陷报告, 以建立一个过程, 避免磁盘空间问题的发生。

我检查电子邮件, 看到了来自缺陷数据库的通知。通知复制给我, 缺陷标记为 DiRT2011 的一部分。和该标签相关的所有细节都在各方的监控之下, 确保在接下来的几个月中他们会注意到。我在等待恢复完成的同时修复了第一个缺陷。

第二个缺陷需要花费更多的时间。我们需要在季度资源估算和分配过程中增加恢复区域。而且我们将在监控系统中添加一些规则, 以检测数据库大小是否接近恢复区域的大小。

Tom: 好了, 服务备份已经读取。我将运行服务的克隆, 并发送给你一条即时消息, 包含用于访问它的 URL。

[几次击键之后]

Rosanne: 好了, 我可以访问数据, 看起来不错, 祝贺你!

Tom: 谢谢!

Rosanne: 好, 你可以继续工作了。哦, 我可能不应该告诉你, 测试控制人说下午 2 点会有些乐子。

Tom: 你知道我的班次在下午 3 点结束, 对吗? 如果你恰巧延迟了一个小时……

Rosanne: 没有那么好运气。我在加利福尼亚, 在你所在时区的下午 3 点, 我会去吃午饭。

演练结束之后一分钟, 我接收到一个电子邮件, 包含指向演练后文档的链接。我用发生的情况更新了文档并提供指向所提交缺陷的链接。我还想到了改进过程的其他几种方法并记入文档, 在我们的缺陷数据库中为它们提交了功能请求。

下午2时,我的传呼机并没有响,但是仪表盘上看到在佐治亚州发生了运行中断。内部聊天室中每个人都在聊这件事。我并没有太担心。我们的服务运行于世界各地的4个数据中心,系统已经自动地将Web请求重定位到其他3个位置。这种迁移没有漏洞,只会丢失“在途”的查询,这还在SLA的范围之中。

我的收件箱出现了一个新邮件,解释僵尸已经入侵佐治亚州,试图吃掉数据中心技术人员的大脑。僵尸已经切断了到数据中心的网络连接,没有任何进出的网络流量。最后,电子邮件指出这是DiRT演练的一部分,没有任何技术人员的大脑被吞噬,但是网络连接确实被禁用。

[电话再次响起]

Rosanne: 你好!玩得高兴吗?

Tom: 我总是在找乐子,但是我猜你讲的是佐治亚州的运行中断?

Rosanne: 是啊,那些技术人员够丢脸了。

Tom: 我认识他们中的很多人,他们的脑子很大,那些僵尸够吃一阵子的了。

Rosanne: 你的服务仍然在SLA范围内吗?

我看了一下仪表盘,3个数据中心正在处理正常分布到4个位置的工作,延迟略有增大,但是在SLA范围内。实际上,我没有必要观察仪表盘,因为如果延迟不可接受(或者以一定速度增长,如果不检查将达到不可接受的水平),就会收到传呼。

Tom: 一切正常。

Rosanne: 很好,因为我还要监督另一次测试。

Tom: 一群僵尸还不够?

Rosanne: 依我看还不够。你知道的,SLA规定你的服务应该能够在同时有两个数据中心停运的情况下存活。

她说的对。我们公司的标准是可以在同时有两个数据中心停运的情况下存活。原因很简单。数据中心和服务必须能够偶尔暂停,以进行计划内的维护。在这个时间窗口中,另一个数据中心可能因为计划外的原因(如网络中断或者停电)停运。在两个同时发生的运行中断下存活的能力称作N+2冗余。

Tom: 那么,你希望我做什么?

Rosanne: 假定欧洲的数据中心因为计划中的预防性维护而停机。

我按照规程暂时关闭了欧洲的服务。来自欧洲客户的Web流量分布到剩下的两个数据中心。因为这是有序的关闭,不会丢失任何查询。

Tom: 完成!

Rosanne: 仍然在SLA范围内吗?

我看了看仪表盘,发现延迟进一步增加了。整个服务运行在两个较小的数据中心上。两个停运的数据中心都大于较小的正常工作数据中心的总和,但是仍然有足够的容量处理这种情况。

Tom: 我们刚好能够满足 SLA。

Rosanne: 祝贺你！通过测试了，你可以启动欧洲数据中心的服务了。

无论如何，我还是决定要提交一个缺陷。我们保持在 SLA 范围内，但是太过靠近极限令人不快，当然应该做得更好。

我看着钟，已经快要下午 3 点了。在下一位值班人员就要上线时，我结束了演练后文档的提交。我给她发了一个即时短信，说明她所错过的一场好戏。

我还提醒她，锁好办公室的门，天知道僵尸下一次会袭击哪里。

15.5 事故指挥系统

公共安全领域使用事故指挥系统管理运行中断。IT 运营可以改编这一过程处理运营事故。这一概念最初是因为 Brent Chapman 在其题为《Incident Command for IT: What We Can Learn from the Fire Department》(IT 事故指挥：我们可以从消防部门学到什么，Chapman, 2005) 的谈话而流行起来的。Brent 在 IT 运营和公共安全上都有丰富的经验。

在系统管理的领域之外，来自不同公共安全组织（如消防、警察和医务）的团队一起响应紧急情况和灾难。他们已经开发出事故指挥系统（Incident Command System, ICS），使这种协作可以高效进行，根据情况的变化扩大或者缩小规模。

ICS 的设计是为了建立灵活的框架，人们可以在这一框架下高效地工作。ICS 的关键原则如下：

- ❑ **标准化组织结构：**ICS 团队由事故指挥员和指挥子系统组成：运营、后勤、规划、行政 / 财务和可选的统一指挥部、公共信息官员和联络员。
- ❑ **各方责任的明确定义：**有且只有一个事故指挥员。ICS 团队的每一个人只向一位 ICS 事故主管报告。
- ❑ **明确的授权委托：**事故指挥员建立某些关键分支，如后勤、运营和规划，并将这些职能委派给它们的指挥员。
- ❑ **目标管理：**为事故确定清晰的目标和优先权。告诉人们你想要实现的目标，而不是如何去实现，让他们领会出在当前情况下完成目标的最佳途径。
- ❑ **可以伸缩的有限控制范围：**在 ICS 下，主管不应该有超过 7 个直接报告人。理想情况下，应该有 3~5 人向一位主管报告。随着涉及人员的增长，组织扩张并设立新的主管。通过这种方法，保持对职责的限制。记住，这是为响应特定事件所建立的临时组织结构。不同事件中，同样一组人员可能以不同方式组织。
- ❑ **常见术语和组织框架：**通过使用 ICS 角色和职责作为公共框架，来自不同组织的不同团队可以清晰地理解“谁”做“什么”以及他们的职责所在。

ICS 的详细介绍可参见美国联邦应急管理署（FEMA）网站（<http://www.fema.gov/incident-command-system>），FEMA 应急管理学院发布了免费的自学和其他培训材料（<http://training>）。

fema.gov/EMI/)。更容易找到的简介是关于 ICS 的维基百科文章 (http://en.wikipedia.org/wiki/Incident_command_system)。

15.5.1 工作原理：公共安全领域

当公共安全事故开始时，第一项议程是做好组织，首先决定负责人。在事故中，第一位到达的合格响应者自动成为事故指挥员 (Incident Commander, IC)。ICS 团队由事故指挥员和运营、后勤、规划/状态和管理/财务的指挥子系统组成，如图 15.1 所示。还要指定一位公共信息官员，以处理响应团队 (内部和外部) 之外的事事故沟通。联络官是外部机构 (如第三方供应商) 的主联络人。安全官监控安全状况。

所有事故都有一个指挥部 (管理部门)，大部分事故还有一个运营部，直接提供紧急服务，如灭火或者提供医护服务。管理结构是严格的层次结构，在紧急情况下，没有时间应付难以捉摸和低效率的矩阵管理。

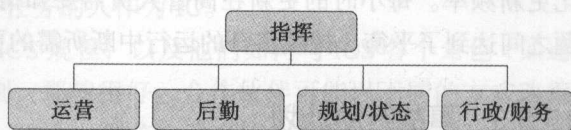


图 15.1 ICS 基本组织结构

IC 到达时，他常常独自一人承担所有角色。随着更多的人到场，IC 按照需要委派任务。新人到来时，他们到行政/财务 (如果存在的话) 部门报道以便签入，然后，IC 将他们安排到某个职能。IC 需要有确定新资源分配最优方式的全局视野。

IC 任务不会只因为更高级别人员或者不同部门到达而转移给任何人。例如，警察局长不会自动地胜过一名警官，消防队员也不会自动胜过护理人员。控制的转移是明确的，由当前的 IC 将控制权移交给另一个明确接受任务的人。任务的移交会造成混乱，只能在必要时进行。

团队保持小的规模。3~5 人的团队规模被认为是最优的。如果子指挥部受到 7 份直接报告，该团队应该在加入下一个人时分割为两个。

15.5.2 工作原理：IT 运营领域

将上述系统改编为 IT 运营组织，运营团队处理事故的运营方面——就像真正的救火。后勤团队处理资源，如人和材料，确保运营部门有实施工作所需的资源。规划团队负责预测情况和资源需求，收集和显示事故相关信息。行政/财务团队处理一般的行政支持和预算。

最初，IC 是第一个到场的人，在 IT 中通常是响应警报的值班人员。不过，因为这个人已经投入与警报相关的检修工作，对他来说，发出警报并将 IC 任务转移给下一个合格的响应人员，而自身继续作为运营 (Ops) 领导人更有意义。IC 任务在换班时明确移交，或者在 IC 感到疲劳、需要休息时移交。这里的关键是，IC 任务的移交往往作为某个经过深思熟虑的过程的一部分，而不是在新人出现时自动进行。如果更有资格的人到场，当前 IC 可以决

定是否值得进行切换 IC 的破坏性移交。

对于小的 IT 事故，IC 自行处理所有领导任务。但是，当事故规模增大时，由于更多的同事注意到运行中断，或者 IC 联系其他同事，所涉及的人员更多。随着人们的出现，IC 为他们分配任务，在必要时建立子团队。值得注意的是，IC 和 Ops 领导人的角色应该尽快分开。IC 有全局视野，而 Ops 领导亲临第一线，直接处理事故。试图同时处理这两个任务往往造成满盘皆输。

长时间的运行中断需要频繁地向管理层和其他利益相关方提交状态更新。指定一个人作为公共信息官，IC 就可以不被高管人员的更新要求或者用户关于“机器正常了没有”的问题分散精力。每次状态更新结束时都应该说明下一次状态更新的预期时间。许多组织标准化更新频率。每小时的更新在高管人员需要知情和技术工作人员需要专注于手上的技术问题之间达到了平衡。持续多日的运行中断所需的更新频率更低，以便避免重复。

15.5.3 事故行动计划

ICS 中会创建事故行动计划（Incident Action Plan，IAP），并在事故期间持续求精。这个 1~2 页的文档回答 4 个问题：

- ❑ 我们应该做什么？
- ❑ 谁负责这些工作？
- ❑ 我们如何相互沟通？
- ❑ 如果有人无法响应，采用什么规程处理？

在 IT 运营中，我们加入第 5 个问题：

- ❑ 如果检测到更多故障，采用什么规程处理？

创建一个模板，供 IC 用于制作这个文档。将模板放在易于访问的标准位置，让每个人都想知道在哪里能找到它。图 15.2 中有一个示例。

我们应该做什么？

主 Web 服务器下线。我们必须恢复其全部功能。

谁负责这项工作

IC 是 Mary。SRE 团队在开发人员 Rosanne、Eddie 和 Frances 帮助下负责此项任务。

我们如何相互沟通？

视频聊天室：

https://plus.google.com/hangouts/_/example.com/outage

如果有人无法响应，采用哪种规程？

如果可能，他们应该通过电话（201-555-1212）、短消息、即时消息或者视频聊天室报告给 IC。如果无法做到或者有人怀疑其他人已经无法响应，将此信息报告给 IC。

如果检测到更多故障，采用哪种规程？

IC 将负责领导筛选工作、评估情况和决定如何推进。

图 15.2 事故行动计划（IAP）样本

15.5.4 最佳实践

为了确保成功,每个人应该使用 ICS 规程作为事故指挥员进行实践,在新雇员入职培训中,应该向他们介绍 ICS 规程,升级合作伙伴应该在 ICS 规程投入使用时得到通知。

团队应该在小事故中使用 ICS 规程进行实践,使每个人在严重的情况发生时熟悉这一过程。在处理非紧急情况(如数据中心迁移和重大升级)和安全事故、怀有敌意的解雇人员时也应该使用 ICS 规程。

ICS 培训应该作为运营团队新雇员入职过程的一部分。每个人都理解术语和过程,保证系统运转良好,是十分重要的。

团队成员应该轮流担任事故指挥员,以了解团队一起工作时的范围和能力。在非紧急情况下应该选择在最长时间没有担任这一任务的人作为 IC。

应该通知升级合作伙伴你的团队使用 ICS 规程,以及他们如何与 ICS 各个角色(如通信官)互动。如果他们内部也使用 ICS 规则,那就很好。合作伙伴可能以不同的方式改编 ICS 规程用于 IT,这可能导致混乱和冲突。因此,应该实践多团队事故,以找到和解决问题。可以使用模拟问题和实际问题,以及非紧急情况问题。筛选涉及两个团队的重大缺陷时是这类实践的好机会。

15.5.5 ICS 示例

举个 ICS 规程使用的例子,假定一次重大的运行中断正在影响 XYZ 公司的 Web 运营。运行中断时值班的响应者 Bob 启动 ICS 过程,作为临时事故指挥员。Bob 所做的第一件事是向他的团队通知运行中断的有关情况,并将事故指挥员的任务移交给团队领导人 Janet。Janet 立刻开始组建 ICS 团队,将运营任务委派给 Bob。她将运行中断的有关情况告知自己的经理 Sandy,请求他担任公共信息官。

Bob 和 Janet 描述他们所认定的情况和所要做的工作,启动了事故行动计划。Bob 采取行动,Janet 协调工作。Bob 解释道,他需要某些资源(如故障切换机器),团队可能在稍后进行这项工作。作为事故指挥员,Janet 联系一些人执行后勤任务。她选择 Peter,并指导他从 Bob 那里了解需求。这位新的后勤部主管指派人员得到运营部门需要的机器资源,并安排在下午六点送到。

Janet 指定监控小组的某个人,要求得到运行中断的详细信息。Jyoti 同意担任规划部主管,开始和监控小组的其他成员一起收集请求的信息。他们忙于满足预测的要求,如了解到 Web 服务目前在欧洲的请求量达到每日的高峰,在北美的请求量也正在上升。

随着运行中断的持续,Janet 的经理 Sandy(公共信息官)与 Janet(事故指挥员)、Bob(运营)、Jyoti(规划)和 Peter(后勤)接口,跟踪发生的情况,并向人们通报事故的进展。Sandy 委派一名团队成员更新 XYZ 公司客户所用的状态页面和公司内部的状态页面。

与此同时,运营部门的 Bob 已经确定某个数据库需要故障切换,更新副本。按照目标管理,他要求后勤部门提供另一个资源以完成副本和负载平衡工作。后勤部门找到公司 IT

部门的另一位在副本系统上有经验的同事，授权该同事在运行中断期间协助 Bob。Bob 开始数据库故障切换，他的新助手则开始所需的负载平衡和副本工作。

新的副本完成初始复制并开始为请求提供服务。Janet 和 ICS 各部门主管确认服务状态已经恢复正常。这一事件宣告解决，ICS 过程明确终止。Janet 记录行动项目，领导事后剖析工作。

15.6 小结

为了很好地处理重大运行中断和灾难，我们必须准备和实践。无知可能是天赐之福，但是实践才会进步。通过在受控环境中测试发现灾难恢复过程不完整，比在真正的紧急状况下意外发现问题更好。

为了在每个级别上都做好准备，需要在个人、团队和组织级别上实践灾难恢复技术的策略。每个级别依赖于前一级别达到的能力。

灾祸之轮是通过常见和不那么常见的灾难情景训练个人的游戏。应急演练是练习特定过程的实际测试。应急演练应该由同一个人在过程上反复进行，直到过程有效且可以顺利执行。然后，由团队每个成员执行过程，直到每个人对自己执行任务的能力建立信心。

关闭随机选择的机器或者服务器的测试可以发现未经测试的故障情景。这些测试可以在指定时间里进行，也可以作为生产的一部分持续进行，以确认系统对故障的弹性没有退化。

游戏日或者 DiRT 演练是组织范围的测试，用于找出涉及多个团队的过程中的不足。这些演练往往持续数天，包括切断主要系统或者数据中心。

DiRT 活动需要大量的计划和协调，以降低风险。测试应该由中央规划委员会根据特质、影响和风险加以批准。有些测试由单独团队发起，影响全公司范围的测试则由中央委员会规划。

消防、警察、医疗和其他公共安全组织采用事故指挥系统（ICS）协调灾难和其他紧急情况中的工作。ICS 提供了标准化的组织结构、过程和术语。这一系统可以改编用于协调重大 IT 运行中断的恢复。为了保持实践，团队应该对较小的事故使用 ICS 规程。

练习

1. 什么是反脆弱系统？和脆弱系统及弹性系统有何不同？
2. 将应急演练作为可靠性测试的一部分有哪些理由？
3. 画出表示 15.1.1 节中讨论的故障理想频率的图表。
4. 描述你的环境中可以从可靠性测试得益的关键子系统，并解释测试的方式。
5. 对比对故障的传统态度和 DevOps 对故障的态度。可以在你的环境中应用哪些教训？

监控基础知识

只要注意，就能观察到很多。

——Yogi Berra

监控是我们获得所运行系统可见性的主要手段。这是观察用于短期和长期决策的事物状态信息的过程。监控的运营目标是检测运行中断的先兆，以便在真正停机之前解决问题。收集有助于未来决策的信息，以及检测真正的运行中断。监控是一个难题，组织往往监控错误的事物，有时候没有监控重要的事物。

理想的监控系统使运营团队无所不知、无所不在，想一想，有了根密码我们就无所不能，这时我们简直是全能的统治者。

分布式系统很复杂，无所不知、无所不晓意味着我们的监控系统应该提供系统的可见性，找出完成工作必需的所有信息。我们可能不知道监控系统所知的一切，但是在需要的时候可以找到。分布式系统太大，任何人都无法知晓其中发生的一切。

分布式系统的规模很大，这意味着我们必须无所不在，同时存在于每个地方。监控系统使我们在系统分布到全球时仍能做到这一点。在传统系统中，人们可以想象系统管理员对系统有足够的了解，可以关注所有重要组件。不管这种感觉是否正确，我们知道在分布式系统中绝对不是这样。

分布式计算中的监控与企业计算的监控不同，监控不仅是在服务或者网站停顿时将你叫醒的系统。理想情况下，应该绝不会发生这种情况。选择包含运行中断响应的策略，就意味着我们选择了“内含”运行中断的运营策略。我们可以改进响应运行中断的速度，但是运行中断仍然会发生，那不是运行可靠系统的方法。

必知术语

服务器：提供功能或者 API 的软件（不是硬件设备）。

服务：由许多服务器组成的用户可见系统或者产品。

机器：虚拟或者物理机器。

QPS：每秒查询数量。通常是每秒接受的 Web 点击数量或者 API 调用数量。

昼夜循环：白天较高、夜晚较低的指标。

相反，监控应该设计为及时检测运行中断的先兆，以避免问题的发生。系统应该得到测量和监控，以实现上述策略，这比检测运行中断更难，但是更有用。

16.1 概述

要理解监控，必须首先理解其特定的术语。

计量（Measurement）指的是描述系统某个方面的单一数据点，通常是可以进行数值运算的一个值，例如 5、-25、0 或者空（Null）。计量也可以是字符串，例如，当前挂载文件系统的版本号或者逗号分隔列表。

指标（Metric）是具有名称和时间戳的计量。例如：

```
spellcheck:server5.demo.com:request-count@20140214T100000Z = 9566
```

不同监控系统使用不同格式标记数据。在上面的例子中，我们有来自运行于 server5 上拼写检查服务器的一个指标。计量是请求计数，可能是从开始起服务器接收到的请求数量。在 @symbol 之后的是日期戳记。计量值是 9566。

其他指标示例包括在用 RAM 总数、队列中未处理请求的数量、运行代码的构建版本、未解决缺陷数量、缺陷数量和 Amazon AWS 上的总成本。指标还有相关联的元数据，特别是，数字值通常有一个相关的单位。知道单位可以进行自动化转换、设置图表标签等。许多监控系统不跟踪指标的单位，需要人们根据指标名称或者其他背景信息猜测，人工执行转换。这一过程很容易出错。

计量频率（Measurement Frequency）是新计量进行的速率。不同指标以不同频率收集。每 5 分钟都会收集很多数据，但是有些指标每秒都收集许多次，其他则每天或者每周收集一次。

监控视角（Perspective）是监控应用程序收集计量的位置。视角的重要性取决于指标。不管采用什么视角，某个接口上发送的数据包数都一样。相反，页面加载时间和其他定时数据取决于视角。人们可以从全世界的不同位置收集相同计量，了解距离对页面加载时间的影响，或者检测特定国家存在的问题或者 ISP 的连接性问题。另外，如果视角不如一致性重要，计量可以直接在一台机器上收集。

收集来自许多视角信息的一种方法是**真实用户监控（Real User Monitoring, RUM）**。RUM 从 Web 浏览器收集实际应用性能计量。代码被插入网页，收集指标并传回发送原始

网页的网站。

16.1.1 使用监控

监控不只是收集数据，它还可以多种方式使用，这些使用都有特定的术语。

可视化（Visualization）是将多个计量吸收到一个可视化表现中。这些图表可以找出趋势，在系统和时间范围之间做出比较。

趋势（Trend）是一系列计量指标的方向。例如，可以运用可视化了解服务的使用是增长还是收缩。

警报（Alerting）意味着让某人或者另一个系统注意某件事物。QPS 的突然降低会造成向值班人员发出警报。警报通常必须在一段时间内确认。如果超过最后限期，则向另一个人发出警报，这个过程称作**升级（Escalation）**。

可视化有助于更深入了解系统，可以帮助设计、规划和沟通等所有环节。趋势可以用于容量规划（第 18 章中将详加讨论），警报应该用于警告人们可能造成运行中断的情况，还可以作为运行中断发生时的最后警告手段。

16.1.2 服务管理

最后，这里介绍一些偶尔会使用到的服务管理术语，它们大部分来自于商业世界：

- **服务水平指标（Service Level Indicator, SLI）**：关于如何计量的协议。例如，可能定义计量的内容、方法以及视角。
- **服务水平目标（Service Level Target, SLT）**：服务的目标质量。换言之，SLI 的预期最小或者最大值。在 ITIL V3 之前，这被称作服务水平目标（Service Level Objective, SLO）。SLT 的例子之一是某种可用性等级或者最大允许延迟。
- **服务水平协议（Service Level Agreement, SLA）**：规定 SLI、SLT 和未达到 SLT 时罚款的合同。例如，对于超过一小时的运行中断可能有退款或者罚金。SLA 一词往往被滥用，表示 SLT 或者任何服务水平需求。

16.2 监控信息的消费者

监控信息有许多消费者，每个消费者有不同的需求。监控运营健康的人需要立刻知道状态变化，以快速做出反应——他们需要实时监控。容量规划人员、产品经理和其他人需要长期收集的指标以发现趋势——他们需要历史监控数据。

更细致地区分消费者的方法是 Dickson 模型（Dickson, 2013），它使用了 3 个特性：分辨率、延迟和多样性。这些特性可以评定为“高”或者“低”。

分辨率（Resolution）描述指标收集的频率。高（R+）是每秒、分或者小时收集许多次。低（R-）是一天收集许多次。

延迟 (Latency) 描述可以根据信息采取行动之前经过的时间。低 (L+) 是实时响应。高 (L-) 表示数据被存储供以后分析, 可能用于每日、每周或者每月统计。(注意, + 和 - 的使用和 R/D 相反。这种情况下, + 被认为更难设计。)

多样性 (Diversity) 描述收集的指标数量。高 (D+) 表示许多指标, 可能有关于许多不同服务的许多计量。低 (D-) 表示专注于特定或者小的指标集。

消费者可以用一个三元组表示。例如, (R+, L-, D+) 描述高分辨率、高延迟、高多样性消费者。

按照这些坐标轴, 我们可以描述监控信息的主要用户:

运营健康 / 响应 (OH) (R+, L+, D+)	高分辨率, 低延迟, 高多样性。系统健康, 我们接到相关的传呼
质量保证 / SLA (QA) (R+, L-, D+)	高分辨率, 高延迟, 高多样性。抖动、延迟和其他质量相关因素的较长期分析
容量规划 (CP) (R-, L-, D+)	低分辨率, 高延迟, 高多样性。预测和采购更多资源
产品管理 (PM) (R-, L-, D-)	低分辨率, 高延迟, 低多样性。确定用户数量、成本和其他资源

运营健康是典型的监控, 检测异常情况并生成警报, 是最高要求的用例。分辨率和延迟必须足以发现问题并在 SLA 范围内响应。这通常要求访问最新的所有指标, 用于高速分析的实时计算以及可靠的警报。存储系统必须同时兼具高速和大容量。实际上, 这种监控给监控基础设施的各个部分带来压力。

质量保证通常涉及特定质量指标 (如可变性) 的中长期分析。例如, 有些查询应该始终花费大约相同的时间, 变动很小。质量保证检测这种可变性, 就像汽车装配线质量保证团队寻找所制造产品中不合格品和不可接受变化一样。因此, 质量保证需要高分辨率数据, 但是延迟并不需要, 因为数据往往是在事后批量处理的。

质量保证还包括寻找和修复缺陷时必需的信息, 如调试日志、进程跟踪、堆栈跟踪、核心转储和分析工具输出。

容量规划 (Capacity Planning, CP) 是预测未来资源需求的过程, 这些预测需要粗略的指标, 如当前机器数量、已使用的网络带宽数量、每用户成本和机器利用率及效率以及资源将耗尽时的警报。CP 还关心产品变化时资源使用的变化, 例如, 新发行版本是否需要明显不同的资源。CP 是第 18 章的主题。

产品管理 (Product Management, PM) 需要用于计算转化率、用户计数和用户留存分析 (常常称作 7 日活动或者 30 日活动) 等关键绩效指标 (Key Performance Indicator, KPI) 的超低分辨率数据。PM 得益于大量历史数据的长期视图和可视化, 帮助他们了解趋势。

遗漏的 (R, L, D) 组合

认真观察的读者会注意到，并不是所有 R、L 和 D 的组合都出现在模型中。有些组合确实描述了其他运营相关职能。(R+, L+, D-) 是负载均衡器确定是否应该向特定后端发送流量时的需求。(R+, L-, D-) 覆盖了定期批量进行的日志分析。剩下的 (R-, L+, D-) 和 (R-, L+, D+) 我们还没有注意到使用。

16.3 监控的内容

每个组织所应该监控的内容都不同。一般的策略是从企业的 KPI 开始，收集相关的计量。

下面是 KPI 的一些例子：

- ❑ 可用性：网站正常运行时间达到 99.99%（从数据中心外计量）。
- ❑ 延迟：90% 的主页延迟应该不超过 400 毫秒（请求到显示的时间）。
- ❑ 紧急缺陷计数：严重性为 0 的未解决缺陷应该不超过 n 个。
- ❑ 紧急缺陷解决：所有严重性为 0 的缺陷应该在 48 小时内解决。
- ❑ 重大缺陷解决：所有严重性为 1 的缺陷应该在 10 天内解决。
- ❑ 后端服务器稳定性：返回 HTTP 5xx 服务器错误的查询不应超过 n%。
- ❑ 用户满意度：放弃的购物车应该少于 n%。
- ❑ 购物车规模：每个订单购物车中商品数量的中位数应该为 n。
- ❑ 财务：这个月的总收入为 n。

如果不知道组织的 KPI，立刻停下来找出它们。如果没有任何 KPI，那就准备好你的简历，因为公司就像一艘没有舵手的航船，你也可以宣布自己为船长，自己发明 KPI。

必须对系统进行足够的测量，以便在出现故障时发现。为了改进 KPI，我们不仅仅应该计量最终结果。通过更深入地计量过程中的指标，可以做出实现 KPI 的更好决策。例如，为了改进可用性，必须计量成分系统的可用性，得出最终的可用性统计数字。可能有些系统超载、队列太长或者开始出现瓶颈。当树平衡时，搜索树效能最佳。通过监控树的平衡，我们可以将性能问题与树不平衡的时间关联。为了确定购物车被放弃的原因，我们必须知道购物经历中网页是否有问题。

前面提到的所有 KPI 都可以通过选择合适的指标监控，有时候这些指标和其他指标相结合。例如，确定第 90 个百分位数需要一些计算。计算每个订单的平均商品数可能需要两个指标——商品总数和订单总数。

诊断 (Diagnostic) 是收集用于协助调试和性能调整等技术过程的指标。这些指标不一定和 KPI 相关联。例如，我们可能收集一个指标，帮助我们调试间歇性出现但是难以寻找的技术问题。通常有一个从所有机器收集的最小指标集：和 CPU、网络带宽、磁盘空间、

磁盘访问等相关的系统指标。保持一致性便于管理，为每台机器手工制作一份预定的列表不能很好地利用你的时间。

在我们的行业中，“监控一切”是常识，这是为了避免突然发现需要特定指标的历史数据的情况。如果真的照此办理，指标收集可能淹没被监控的系统或者监控系统本身。通过首先关注 KPI，并在必要的时候关注诊断指标，以找到平衡。

最小监控问题

常见的一个面试问题是：“如果你只能监控 Web 服务器的 3 个方面，你会选择哪些？”这是对技术知识和逻辑思维的极好测试，要求你利用技术知识找出许多可能出现问题的“代理”指标。

例如，执行 HTTPS GET 命令可以了解很多情况：我们知道服务器是否启动、服务是否超载、网络是否拥塞。TCP 计时表示第一个字节的传输时间和整个载荷的传输时间。分析 SSL 事务可以监控 SSL 证书有效性和到期。其他两个指标可用于区分这些问题。了解 CPU 利用率有助于区分网络拥塞和系统超载。监控空闲磁盘空间的数量可以找出失控进程、日志填满磁盘以及许多其他问题。

我们建议回击面试官，提出自始至终都仅计量一个指标。假定是一个电子商务网站，只需要计量收入，如果收入以不符合某个时刻特性的方式急剧下降，则说明该网站超载。如果停止产生收入，说明网站下线（如果没有下线，有理由进行调查）。如果急剧上升，我们就知道容量很快会耗尽。这是将所有因素联系起来的一个 KPI。

16.4 留存期

留存期（Retention）是收集的指标数据保存的时长。留存期满之后，旧的指标数据到期、降低采样率或者从存储系统中删除。

监控数据保留的时长对每个组织和服务都不相同。人们通常渴望永远保存所有指标，这能够避免突然发现想要的数据被删除的情况，也比决定每个指标的存储时间更简单。

遗憾的是，永远存储数据是有代价的——不仅是存储硬件，还涉及备份、电源和复杂度。存储足够的数据，在多个存储系统之间分割数据是很复杂的。关于数据保留时长还可能涉及法律问题。

建立留存策略应该从收集业务需求和目标开始，然后将它们转化为存储系统的需求。

一般认为两年是最低的存储期，因为这可以实现年度比较——多多益善。下一个监控系统可能无法读取前一个系统收集的数据，而且很有可能不存在转换过程。在这种情况下，如果每 5-6 年建立新的监控系统，那个时间就是留存的上限。但是时间序列数据库越来越标准化和易于转换，这个上限可能会消失。

我们刚刚开始理解以完整的分辨率保留数十年监控数据的能力的好处。例如，Google

的论文《Failure Trends in a Large Disk Drive Population》（大型磁盘驱动器群的故障趋势，Pinheiro、Weber & Barroso 2007）打破了许多关于硬盘可靠性的神话，因为作者们访问了数十万块硬盘自监控设施（SMART）5年多收集的高分辨率监控数据。当然，不是每个人都有无限的磁盘存储设施，需要某种合并或者压缩。

最简单的合并是删除不再需要的数据。最初可能收集许多指标，但是其中许多都已经变得不相关或者不必要。在建立系统时，收集数据过多总比想要数据时却发现没有收集更好一些。在服务运行一段时间之后，某些指标可能确实没有必要，或者只在短期有用。例如，一些特殊的 CPU 相关指标在调试当前问题时有用，但是一年之后它的效能消失了。

减少存储需求的另一种方法是通过摘要或者降低采样率。利用这种技术，当前数据保持完整，旧数据则被平均数或者其他形式的摘要代替。例如，指标可能按照 1 分钟或者 5 分钟的间隔收集。当数据留存超过 13 个月时，计算每小时的平均数、百分位数、最大值和最小值，并删除原始数据。当数据更旧时（如 25~37 个月），计算 4 小时甚至每天的摘要，进一步降低存储需求。同样，摘要的数量取决于业务需求。如果你只需要知道大约的带宽利用率，每日的值可能就足够了。

16.5 元监控

对监控系统的监控称作元监控（Meta-Monitoring）。如何知道今天没有得到警报的原因是因为一切正常，还是监控系统出现故障？元监控检测监控系统本身出现问题的情况。

监控系统需要有比被监控的服务更高的可用性和可伸缩性。每个监控系统应该有某种元监控。即使最小的系统也需要简单的检查，以确保其仍然运行。较大型的系统应该被监控和任何其他服务相同的伸缩性和容量问题，以避免磁盘空间、CPU 和网络容量成为限制因素。收集数据的准确性和精度应该受到监控，人们还应该监控收集计量指标的失败频度。信息显示必须快于普通人的注意力持续时间。应该监控用于计算 KPI 的数据新鲜度，了解用于计算 KPI 的数据年龄可能和 KPI 本身一样重要。了解 KPI 延迟的趋势，对于了解监控系统的健康状况很重要。

元监控技术之一是部署第二个监控系统，监控主系统。这个系统的共同依赖性应该尽可能少。如果使用不同的软件，同一个缺陷就不太可能导致两个系统同时下线。但是，这种技术增加了复杂度，需要更多的培训，而且必须进行维护。

另一种技术是将网络划分为两个部分，每个部分都使用自己的监控系统。这两个监控系统同时相互监控。例如，具有两个数据中心的站点可以在每个数据中心里部署不同的监控系统，以监控本地机器。这样做节约了数据中心之间的带宽，消除了网络互联这个故障源。

对于一个以上的数据中心，可以使用类似的布置，数据中心两两相互监控，或者每个

监控系统监控大型环路中的另一个系统。

16.6 日志

获得系统可见性的另一种手段是分析日志数据。虽然和监控不直接相关，但是我们在提及这一能力，是因为它所带来的可见性。日志有许多种类：

- **Web “点击” 日志：**Web 服务器通常记录每次 HTTP 访问，以及关于性能的统计数字、访问来源和访问是否成功、错误、重定向，等等。这些数据可以用于多种业务目的：确定页面生成时间、分析用户来源、跟踪用户在系统中的访问路径等。技术运营部门可以使用这些数据分析和改进页面加载时间和延迟。
- **API 日志：**每个 API 调用的日志记录工作通常包括保存调用者、输入参数和输出结果（往往用简单的成功或者错误编码摘要表示）。API 日志对记账、安全取证和功能使用模式很有用。打算淘汰某些过时 API 调用的团队可以使用日志确定受影响的用户。
- **系统日志：**操作系统内核、设备和系统服务都会生成系统日志。这对跟踪硬件问题和系统更改往往很有用。
- **应用程序日志：**每个服务或者服务器生成操作日志。这对于研究错误和调试问题有用，而且提供哪些功能最常使用等业务信息。
- **应用程序调试日志：**应用程序常常在单独的日志中生成调试信息。这类日志往往更详细，但是保留的时间较短。这些日志可用于开发人员和运营人员调试问题。

16.6.1 方法

日志不直接适用 Dickson 模型，如果使用该模型，那么它们应该是 (R+, L-, D-)，因为它们是高分辨率（通常每次操作生成一个日志项）、高延迟（往往在很久以后才批量处理）和低多样性的（通常为特定主题收集，如 Web 服务器点击）。

日志处理系统的架构类似于下一章讨论的监控架构。日志通常从机器和服务中收集并保存在一个中心位置供存储和分析。留存期往往是一个法律问题，因为日志数据往往包含受到监管的个人信息。日志分析工具的市场逐年增长，新的分析方法不断发明。分析 Web 日志可以确定用户进入系统的路径。因此，人们可以找出用户界面中阻碍用户的“死胡同”。应用程序日志可用于找出销售过程中的漏洞，或者识别其他方法无法发现的高价值客户。可以分析系统日志，找出异常情况，甚至预测硬件故障。

日志数据的合并可能相当复杂，因为不同系统生成的日志格式不同。最好是为所有系统确定单一的日志格式，通过提供生成遵循该格式日志的软件库，可以使最小阻力路径（使用程序库）产生所需的行为（遵循标准）。

16.6.2 时间戳

在所有系统中以相同方式记录时间戳，使日志更容易合并和比较。特别是，所有机器应该使用 NTP 或者等价服务保持时钟同步，时间戳应该保存为 UTC 而不是本地时间。

这一点很重要，因为在调试时或者运行中断后查错时，往往比较来自不同系统的日志。例如，撰写事后剖析报告时，人们通过收集来自不同机器和服务的日志（包括聊天室脚本、即时消息会话和电子邮件讨论）建立事件时间轴。如果每个服务记录本地时区的时间戳，仅仅找出事件发生顺序就可能成为一个难题，尤其是在这些系统没有指出所用时区的情况下。

大规模合并日志通常是自动化进行的，合并过程要将所有日志规范化为 UTC。遗憾的是，配置错误可能造成日志数据规范化错误，注意到的时候总是为时已晚。这一问题可以通过在所有日志中使用 UTC 避免。

Google 的日志时间戳使用美国 / 太平洋时区，Tom 在那里工作时深受困扰。这个时区的夏令时日程与欧洲不同，根据年份，每年都有两周的日志规范化格外复杂。这还意味着，必须编写程序，理解每年有一天少一个小时，另一天多一个小时。据说，简单地使用美国 / 太平洋时区是因为第一位 Google 系统管理员没有考虑决策的涉及面。这一时区深深地嵌入 Google 的许多系统之中，没有希望改变。

16.7 小结

监控是获得所运行系统可见性的主要手段，包括用于向我们发出警报，通知需要注意的情况的实时监控，以及方便趋势分析的长期或者历史数据的收集。分布式系统很复杂，需要大规模监控。没有一个人能够独立监控系统，或者凭直觉知道发生了什么情况。

监控的目标是在问题导致运行中断之前发现它们，而不是检测运行中断。如果简单地检测运行中断，运营过程本身就包含了停机时间。

计量是一个数据点，指的是描述系统某个特征的单数据点，通常是一个数字值或者字符串。指标是具有名称和时间戳的计量。

决定监控内容应该从一个自上而下的过程开始。识别企业的关键绩效指标 (KPI)，然后确定收集哪些指标以创建这些 KPI。

监控对于分布式系统特别重要。通过测量系统和服务器和自动收集输出指标，我们可以成为利益相关方所认为的无所不知、无所不在和无所不能的系统管理人员。

练习

1. 监控的目标是什么？
2. 为什么监控在分布式系统中很重要？

3. 为什么运营人员应该无所不知、无所不在、无所不能？为什么这些特性很重要？
4. 计量和指标有什么区别？
5. 如果你只能监控网站的 3 个方面，它们应该是什么？证明你的答案。
6. 16.2 节提到没有观察到 (R-, L+, D-) 和 (R-, L+, D+) 的情况。为什么会这样？
7. 大部分监控系统向设备发出 ping 指令以确定它们是否启动。本章没有提到 ping，根据在本章中学到的知识，向设备发出 ping 为什么是不充分或者失败的运营监控策略？

17.1 传感器与测量

超文本传输协议

传感器和测量组件收集计量。根据对使用的内部组件的知识量，计量可以分为黑盒法和白盒法。计量根据每个项目是单独计算还是定期读取全部并取平均值。计量可以是直接或者合成的。我们可以监控特定操作发生的速度，或者系统允许操作发生的能力。系统可以增加机制以提供仪表（如 CPU 利用率），也可以提供计数器（如某一事件发生的次数）。下面我们更加详细地介绍以上各个概念。

17.1.1 黑盒与白盒监控

黑盒 (Blackbox) 监控意味着计量系统尝试测量系统内部信息，例如系统资源、安全策略等的黑盒。用户不知道系统内部工作原理，只检查系统的外部属性。他们可能猜测内部的相关信息，但是不能确定。换言之，计量是在高度抽象下进行的。

对网站主页发出 HTTPS 请求并测量响应时间。关于响应时间，我们可以在不同条件下进行测量，例如在不同时间、不同网络、不同设备、不同浏览器、不同操作系统、不同 HTML 文档等等。黑盒测试包含试图作为用户进行多步骤过程的监控，例如登录系统 (login)、浏览网页 (browse)、购买商品 (purchase) 等等。测量系统 (Measurement System) 收集计量。

获取特定网页，验证接收的 HTML 是否正在更新。验证方法之一是验证网页中是否包含特定的名人姓名，验证 HTML 中是否包含某些关键字。计量的结果将是 1 或者 0，代表该网页中是否包含某个关键字。这种计量通常用于验证网页是否正在更新。

黑盒监控往往难以提供精确的度量。监控可以反映系统资源使用情况，但不能提供精确的度量。例如，监控 CPU 使用率可以反映系统是否繁忙，但不能提供精确的度量。白盒 (Whitebox) 监控利用了内部知识，例如系统资源使用情况、安全策略等等。这种计量可能提供更精确的度量，但需要更多的内部知识。直接计量 (Direct Measurement) 通常可以直接进行，以达到更高的效率，避免增加距离。

17.1.2 直接计量与合成计量

有些计量是直接计量的，而其他计量则是合成计量的。例如，监控 CPU 使用率是直接计量的，而监控网页响应时间则是合成计量的。

监控架构与实践

什么是犬儒？就是知道所有东西的价格，却对价值一无所知的人。

——奥斯卡·王尔德

本章讨论监控系统的剖析，顺便说明每个组件的最佳实践。

监控系统有许多不同的部件。Dickson（2013）将监控系统分解为图 17.1 中描述的功能组件。计量通过多个步骤组成的管道流动。每一步从配置库中接受配置，使用存储系统读写指标和结果。

传感和计量系统（Sensing and Measurement System）收集计量。收集系统（Collection System）将计量传送给存储系统（Storage System）。从那里，一个或者多个分析系统（Analysis System）从原始数据中提取含义，例如，检测服务器停机等问题或者某个系统指标明显不同于其他系统等异常现象。**警报和升级系统（Alerting and Escalation System）**将这些条件传达给感兴趣的各方。可视化系统显示数据供人们解读。每个组件都通过**配置（Configuration）**库中的信息得知工作的进行方式。

多年以来，已经出现了许多监控产品，既有商业化产品也有开源产品。每种产品似乎都在两个或者三个组件上做得很好，但是

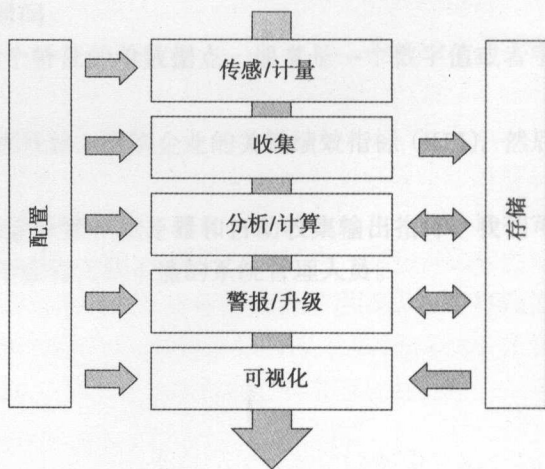


图 17.1 监控系统组件

在其他组件上尚有很多改进之处。有些系统在许多组件上表现出色，但是没有一个能做到尽善尽美。组件之间的接口正在开始标准化，使我们可以混合和匹配组件。这使得新系统可以在不重新开发所有部件的情况下创建，加速了创新。

下面将更深入讨论每个组件、组件的目的以及我们发现的最有用的功能。

17.1 传感与计量

传感和计量组件收集计量。根据对使用的内部构件的知识量，计量可以分为黑盒法和白盒法。计量根据每个项目是单独计算还是定期读取全部并取平均值，计量可以是直接或者合成的。我们可以监控特定操作发生的速度，或者系统允许操作发生的能力。系统可以增加机制以提供仪表（如 CPU 利用率），也可以提供计数器（如某一事件发生的次数）。下面我们更加详细地介绍以上各个概念。

17.1.1 黑盒与白盒监控

黑盒（Blackbox）监控意味着计量系统试图模拟用户，它们将系统当成一个内容未知的黑盒子。用户不知道系统的内部工作原理，只检查系统的外部属性。他们可能猜测内部的有关信息，但是不能确定。换言之，计量是在高度抽象下进行的。

对网站主页发出 HTTPS GET 指令是黑盒监控的一个例子。这种计量不知道任何负载均衡基础设施、内部服务器架构或者所使用的技术。但是，从这一次计量中，我们可以确定许多事情：网站是否启动、响应的速度是多少、SSL 证书是否到期、系统是否产生有效的 HTML 文档等等。黑盒测试包含试图作为用户进行多步骤过程的监控，例如验证购买过程是否正常工作。

获取特定网页，验证接收的 HTML 是否正确，是很有诱惑力的。但是，监控系统必须在每次网站更改时更新。替代方法之一是验证页面包含特定信息。例如，搜索引擎定期查询特定的名人的姓名，验证 HTML 中是否提到这个名人的粉丝俱乐部名称。传达给监控系统的计量将是 1 或者 0，代表该字符串是否找到。

黑盒监控往往高度依赖视角。监控可以从机器本身、从数据中心或者其他故障域内部，或者从世界的不同角落进行。每种测试都回答关于系统的不同问题。

白盒（Whitebox）计量利用了内部知识，因为它是较低级别的抽象。例如，计量可能监控特定 API 调用次数、队列中等待的未决事项数量或者内部进程延迟信息的原始计数器。这种计量可能随着系统内部结构的变化而消失或者改变意义。在白盒监控中，视角比较不重要。这种计量通常尽可能直接进行，以达到更高的效率，避免增加距离造成的数据丢失。

17.1.2 直接计量与合成计量

有些计量是直接的，而其他计量则是合成的。例如，如果每次购买时系统都将收集到

的总金额指标发送给监控系统，这就是直接计量。相反，如果每 5 分钟监控系统计算目前收集到的总金额，这就是合成指标。如果我们所拥有的都是合成指标，就无法反算出组成指标的单独数量。

直接计量是否可行通常是变化频率的函数。在每天只发生一两次购买行为的电子商务网站，直接计量是可能实现的。相反，对于 eBay 或者 Amazon 等繁忙的电子商务网站，如果每次购买都产生一个指标，监控系统可能无法跟上购买的数量。在这种情况下，所能做的就是使用总收入的合成指标。精确的计数是会计和账务系统的任务。

更明显的一个例子是计量在用磁盘空间数量和告知每个数据块的分配和解除分配。每分钟可能发生数百万次磁盘操作，只为了得到在用磁盘空间的指标而收集每次操作的细节是浪费力气。

17.1.3 速率与能力监控

事件频率决定了监控的内容。人们希望知道客户可能购买和正在购买。购买发生的速率决定了监控的对象。如果每天只发生一两次购买，监控客户是否可能购买就很重要，因为购买行为不频繁，我们必须验证购买流程没有出现问题。如果每分钟有数千次购买行为发生，我们通常知道购买可以进行，更重要的是计量速率和趋势。

简言之，速率指标在事件频率很高且存在平滑、可预测趋势时很重要。当频率较低且速率不平均时，能力指标更为重要。

话虽如此，绝不能直接收集速率。速率容易失真，给定两个计数及其时移，我们就能计算速率。给定两个速率及其时移，我们只能推测计数——只有两个计量在精确的时间间隔中收集，且时间间隔与速率的分母同步时才可行。

例如，如果我们计量接收到的 API 调用计数，可能在 12:10:00 收集到指标 10 000，在 12:15:00 收集到指标 16 000。我们可以用计数增量除以时间增量，计算出速率： $(16\,000 - 10\,000) / 300 = 20$ 个 API 调用 / 秒。假定第二次轮询事件被跳过（可能是因为网络问题），下一次轮询事件发生在 12:20:00，收集到计量指标 21 000。我们仍然可以计算速率： $(21\,000 - 10\,000) / 600 = 18.2$ 个 API 调用 / 秒。但是，如果我们收集速率，网络的短暂中断意味着我们无法估算遗漏的 5 分钟内发生了什么。我们可以取两个相邻速率的平均值，但是如果这段时间内出现了一个很大的波峰，我们就不得而知。

17.1.4 仪表和计数器

有些计量是仪表，其他则是计数器。

仪表（Gauge）值是变化的数量，这和现实世界的仪表（如气压表）类似。仪表的例子包括显示可用空闲磁盘空间、温度读数和系统上活动进程数量的指示器。仪表值根据计量的增减而上下变化。

高速网络计数器

频率和潜在变化率决定计数器的大小。如果保存 10 Gbps 网络接口上传输位数的计数器长度为 32 位, 传输位数达到 4 294 967 296 位时计数器将归零。如果接口以 50% 的速度运行, 计数器每 8.5 秒就要归零一次。这样, 除非收集数据的速度快于计数器归零的速度, 否则无法正确地合成接口的流量指标。因此, 变化如此之快的计数器需要 64 位的计数器。

计数器 (Count) 是只会递增的计量, 例如, 服务接收的 API 调用计数或者网络接口上传输的数据包计数。计数器不会递减或者反转。你不能在事后取消数据包的发送或者拒绝 API 调用。但是, 计数器在两种情况下会重置为 0。除非计数器值在持久化存储中, 否则在重新引导或者重新启动时该值会复位为 0。而且, 和汽车里程表在 99999 之后归零一样, 计数器在达到最大值时也会循环为 0。典型的计数器是 16、32 或者 64 位整数, 最大值分别为 65 535、4 294 967 295 和 18 446 744 073 709 551 615 ($2^n - 1$, 其中 n 为位数)。

在计数器读数上进行数学运算相当复杂, 在前一个例子中, 我们简单地将相邻的计数器值相减得出增量。但是, 如果第二个读数小于第一个, 就必须特别小心。

如果计数器循环归零, 两个读数之间的实际增量包含最大值加上新的计数器值: (最大值 $-R_n$) $+ R_{n+1}$ 。但是, 如果计数器因为重启而归零, 准确的增量值就无法得到, 因为我们不知道重启发生时的计数器值。我们可以用高中学过的微积分, 以过去的速率和发生重置的时间做出估算。

确定计数器复位为 0 是因为重启还是因为达到最大值, 是另一个需要推算的问题。如果最后一个读数靠近最大值, 我们可以假定计数器循环到 0。我们同样可以应用微积分提高这一猜测的准确性。

即使使用这么复杂的推算过程, 如果任何一次推算造成不准确的结果, 仍然会造成误差, 有失去准确性的危险。更简单的做法是忽略该问题。对于 64 位计数器, 回绕很少见甚至不存在。对于计数器复位, 如果监控频率足够高, 误差范围将会很小。通常理想的最终结果不是计数本身, 而是变化率。整个时间间隔内的变化率计算不会受到严重影响。例如, 如果有 10 个计数器读数, 会产生 9 个增量, 如果某个增量为负数, 说明发生了一次计数器复位。简单地抛弃负数增量, 用剩余的数据计算变化率就可以了。

Java 计数器

用 Java 编程语言编写的系统使用有符号整数计数器, 有符号整数回绕到负数, 最大值大约是无符号数的一半。

17.2 收集

一旦有了计量, 我们必须将其传输到存储系统。指标从许多位置收集并导入存储系统

供分析。所有传输过程都必须保持指标的一致性。

收集指标的方法有数百种，大部分归入以下两类：推送或者拉取。不考虑推送或者拉取机制的选择，还需要选择数据收集使用的协议。数据收集的另一个方面是服务器本身是直接与收集器通信，还是由外部代理作为服务器及收集器之间的中介。监控系统可能拥有中央收集器或者区域收集器，在数据传递回中央系统之前进行合并。

17.2.1 推送与拉取

推送 (Push) 意味着获得计量的传感器将其传输给收集机制。**拉取 (Pull)** 意味着代理轮询被监控对象并请求和存储数据。

监控有 4 个误区：拉取方法不能伸缩、不应该使用可怕的推送方法、不应该使用可怕的拉取方法、推送方法不能伸缩。打破这些误区很容易。

伸缩性是实现的问题，而不是采用推送或者拉取方法的问题。如果分布均匀，每 5 分钟至少监控 10 000 个项目一次需要每秒连接 33 次。Web 服务器可以处理数百万个人站 TCP 连接。出站连接也没有任何不同。监控系统的优点是可以进行某种优化。它可以在每个 TCP 连接上传输多种计量，可以简单地每台机器创建一个长寿命 TCP 连接。伸缩性是一个问题，但不是不可逾越的障碍。

推送还是拉取更好，是应用的问题。拉取更适合于合成计量（如计数器）。当变化不频繁时，它也适合于直接计量。例如，如果变化每小时只发生几次，每 5 分钟轮询一次可能足以应付了。如果采用可变计量速率（如在诊断问题时以更高频率轮询的系统），拉取方法也更好一些。这种方法类似于提高显微镜的放大倍数。

推送在需要直接观察时更好，如我们需要知道何时发生某事，而不是发生的次数时。在故障时，推送系统更容易保存计量，在可能的时候传输积压的指标。如果需要观察每个离散事件或者规格，那么推送方法更合适。

17.2.2 协议选择

监控中使用许多不同的协议。简单网络管理协议（Simple Network Management Protocol, SNMP）是应该避免的可怕技术。遗憾的是，如果你要处理网络设备，这可能是唯一的选择。如果管理服务 and 机器，则有许多替代品。

大部分替代方法都转向使用通过 HTTP 传送的 JSON。JSON 是基于文本的人类可读数据交换标准。推送采用 HTTP PUT，拉取采用 HTTP GET。因为两种情况下发送的都是同一个 JSON 数据结构，数据处理得到了简化。

SNMP：不是一个简单的管理协议

我们不喜欢 SNMP。大部分实现都以明文传输密码，而且因为每个事务传输一个计量，它的效率低下。该协议的第 3 版可能更安全和快速，但是只有少数供应商采用。

SNMP 协议很复杂，导致其实现总是成为安全相关缺陷的来源。描述该协议的“简单”一词使我们对“复杂”的协议感到忧虑。因为该协议的伸缩性很差，我们有些疑惑，“简单”实际上描述的是不是它所擅长监控的网络类型。名称中的“管理”一词已经伤害了行业，造成网络管理就是收集计量的这一错误认识。

17.2.3 服务器组件与代理、轮询器的对比

收集器可能以不同的方式工作。实施白盒监控的最清晰方法是由运行中的服务器收集自己的计量，处理收集器的结果推送或者拉取。提供这些功能的软件可以纳入某个程序库，方便任何程序使用。例如，每当接收到一个 API 调用，软件可以调用 `metric.increment('api-calls-count')` 递增 `api-calls-count` 计数。为了收集视频服务器中使用的带宽，在每次写入网络时，软件可以调用 `metric.add('bandwidth-out', x)`，其中 `x` 是发送的数据量。程序库在 `bandwidth-out` 的名下维护这个运行总数。整个程序中都使用此类函数计数、加总或者记录计量指标。

如果收集通过推送进行，程序库生成一个线程，定期唤醒以向监控系统推送指标。拉取的实现是由程序库生成一个线程，监听特定 TCP 端口上的轮询事件并用某些或者所有指标响应。如果服务器已经处理 HTTP 请求，可以简单地将库连接到特定路径或者路由。例如，对 `/monitor` 的 HTTP 请求将调用程序库，然后由库以指标的 JSON 表示响应请求。

遗憾的是，我们不是总能修改软件，以上述方式收集指标。在那种情况下，我们在收集数据的机器上运行软件读取日志、状态 API 或者系统参数并拉取或者推送信息。这种软件称作代理 (Agent)。例如，`scollector` 是在 Linux 机器上运行，调用任意数量的插件收集指标，然后推送到 Bosun 监控系统的软件代理。插件可以收集关于操作系统、磁盘系统和许多流行开源系统（如 MySQL 和 HBase）的性能信息。

有时候，我们无法修改软件或者在机器上安装软件。例如，网络交换机、存储区域网络 (Storage Area Network, SAN) 硬件和不间断电源 (Uninterruptible Power Supply, UPS) 等硬件设备不允许用户安装软件。在这些情况下，软件在一台机器上运行，轮询其他设备、收集信息并通过推送或者拉取机制传输到监控系统。这类软件称作轮询器 (poller) 或者监控代理 (Monitoring proxy)。

17.2.4 中心与区域收集器

有些监控系统可以在每个区域布置收集器并将收集到的数据转发到主监控系统，从而扩展到全球范围。这类收集器称作远程监控站 (Remote Monitoring Station) 或者聚合器 (Aggregator)。聚合器可以放在每个数据中心或者地理区域。它可以通过推送或者拉取接收指标，通常合并信息并以更有效率的方式传输给主系统。这种方法可用于将系统扩展到全球，节约每个数据中心之间的带宽。另外，它还能实现系统的纵向扩展，每个聚合器能够

处理一定数量的设备。

17.3 分析和计算

一旦完成数据收集，这些数据就可以使用和解读。分析从原始数据中提取含义，是最重要的组件，因为它首先生成证明监控系统有用的结果。

实时 (Real-time) 分析在数据收集时检查，通常是计算代价最高的分析，保留用于关键任务，例如确定应该发出警报的条件。为了高效地进行分析，监控系统在收集时输出数据，发送一份拷贝到存储系统，另一份拷贝到实时分析系统。存储系统也可以在 RAM 中保存最近收集指标的拷贝。例如，在 RAM 中保留最近一小时的指标，并构造所有警报规则，只引用最后一小时的历史数据，可以高效地处理数百或者数千条警报规则。

典型的实时分析涉及数十或者数百条**警报规则**，这些规则同时处理以找出异常情况（称为**触发器**，Trigger）。触发器的例子包括服务下线、在 x 分钟内服务器的 HTTP 响应超过 n 毫秒或者空闲磁盘空间降到 m 兆字节以下。实时分析系统包含编写描述这些情况公式的语言。

这种分析还可以检测不需要立刻留意、但是如果不予干预可能造成更严重问题的情况。这种情况应该生成单据而非警报。问题的分类参见 14.1.7 节。警报应该保留用于需要立刻注意的问题。触发警报时，提示本章后面描述的警报和升级管理器采取措施。

短期 (Short-Term) 分析检查前一天、周或者月收集的数据，仪表盘通常属于这一类。它们不经常更新，往往几分钟一次或者在有人调用特定网页时按需更新。短期分析通常查询存储指标的磁盘拷贝。近期分析还用于为不太紧急、不需要立刻注意的问题生成单据。参见 14.1.7 节。

仪表盘系统通常包含生成 HTML 页面的模板语言和描述数据图表的语言。数据图表描述在 URL 中编码，使其可以作为嵌入图像包含在 HTML 页面中。例如，URL 可能指定比较特定服务上个月两个指标比率的图表。在计算数百个 Web 服务器的延迟之后，URL 还可以指定 10 个最慢 Web 服务器延迟的直方图。

长期 (Long-Term) 分析通常检查较大时间跨度中收集的数据（往往是某个指标的整个历史），以生成趋势数据。在许多情况下，这包括生成和存储数据摘要（平均值、总和等），以便更快地浏览数据，但是这些数据的分辨率较低。因为这类分析需要大量处理，结果通常永久存储而不是在需要的时候重新生成。有些系统还处理在不同媒介上存储的旧数据的情况——例如磁带。

异常检测 (Anomaly Detection) 是确定特定计量不在预期范围内。例如，检查同类型的所有 Web 服务器，检测是否生成与其他服务器显著不同的指标。这可能暗示某个服务器遇到了其他服务器所没有的困难。异常检测可以找出你没有监控的问题。

异常检测还可以有预测性。可以创建数学模型，使用去年的数据预测今年应该发生的

情况。然后，人们可以检测何时今年的数据与预测有显著的差别。例如，如果可以预测每个国家的 QPS，识别和预测值相差 10% 以上的情况，那可能是检测区域性运行中断还是因为整个南美国家停下来观看特定体育活动的好办法。

实时进行跨越多个系统的异常检测，在计算上可能很困难，但是这种系统越来越普遍。

17.4 警报和升级管理器

警报和升级组件管理当检测到异常情况时与值班人员及其他人员沟通的过程。如果在一段时间内这些人无法联系，该系统会尝试联络其他人。14.1.7 节讨论警报策略和各种沟通技术。

警报组件的第一个任务是引起值班人员或其替代人员的注意，下一项工作是沟通具体信息。前者通常通过传呼机或者短消息完成。由于这些系统只允许发送简短的消息，通常由第二种方法（如电子邮件）传达完整的消息。

这些消息应该传达如下信息：

- **故障情况：**用技术术语和普通语言描述问题。例如，“XYZ 服务的 QPS 过高”是很清晰的，而“错误 42”不清晰。
- **业务影响：**问题的规模和范围，例如，影响的机器或者用户数，服务是否缩减或者完全不可用。
- **升级链：**升级链即联络人，以及联络人无响应时需要联系的人。通常为每个服务或者每组服务定义一个或者两个升级链。
- **建议的解决方案：**解决问题时的简要指南。这最好使用与此警告关联的剧本条目链接，如 14.2.5 节中的描述。

最后两个项目在警报规则创建时可能难以编写。具体的业务影响可能未知，但是你至少知道哪个服务受到影响，可以用这一信息作为占位符。当警报规则被触发时，细节将会变得清晰。花时间记录你的想法，不要丢失关键的信息。

更新影响和分辨率，作为事后剖析工作的一部分。将利益相关方聚集在一起，要求受影响的人用自己的业务术语解释受影响的情况。要求运营利益相关方评估采取的步骤，包括哪些步骤行之有效、哪些步骤还可以改进。将这些信息与剧本中的信息对比，在必要时更新。

17.4.1 警报、升级和确认

警报系统负责将警报提交给合适的人，并在其没有响应时升级到其他人。正如 14.1.5 节的描述，这一信息被编码到值班日程表（Oncall Calendar）中。

在大部分情况下，工作流程包括与主值班人的沟通。他们以“确认”或者“是”等短消息回复、单击链接或者其他手段确认警报。如果一段时间内没有确认，尝试升级列表中

的下一个入。

具备负面确认（“不确认”）警报的能力可以在升级期间节约时间。“不确认”将警报立刻升级到列表中的下一个入。例如，如果值班人员接收警报但是因为互联网连接中断而无法干预问题，他可以简单地不做任何处理，几分钟之内升级自动发生。但是，如果升级时间表每五分钟向值班人员发送传呼，在3次尝试之后才做升级，意味着将行动推迟15分钟。在这种情况下，值班人员可以回复“不确认”，升级将立刻发生。

警报洪泛或者“传呼风暴”——同时发送几十或者几百条警报的情况——是不可避免的。这通常是因为一次网络中断事故造成许多警报规则被触发。在大部分情况下，有一种机制自动抑制相关的警报，尽管组织尽了最大努力，洪泛仍可能发生。因此，警报系统应该有同时确认所有警报的能力。例如，用“所有”或者“闭嘴”等短消息回复，特定人员的所有未决警报和下5分钟接收到的所有警报将被确认。实际的警报可以在警报仪表盘上看到。

有些警报管理器采用两阶段确认。首先，值班人员必须确认接收警报。这确定了解决问题的人，在值班人员工作时禁用特定问题的警报。当问题解决时，值班人员必须“解决”警报，表示问题已经修复。这种系统的好处是更容易生成关于解决问题所需时间的指标。但是如果这个人忘了标记问题解决怎么办？系统必须定期发出提醒警报，这就挫败了两阶段确认的意图。因此，我们感觉两阶段确认提供的实际价值不大。如果希望记录解决问题所花的时间，可以让系统检测何时不再触发警报，这更为准确，也比要求运营人员人工指出问题解决更省心。

17.4.2 静默与抑制

从运营上看，必须能够在任何时候使警报静默。例如，在计划内维护期间，维护人员不需要接收系统下线的警报。依赖于该系统的系统也不应该生成警报，因为从理论上，他们的运营团队已经知道计划内的维护。

处理这种情况的机制称作静默（Silence），有时候称作维护（Maintenance）。静默状态需要指定开始时间、结束时间和需要静默的警报规格。在指定需要静默的警报时，接受通配符或者正则表达式是很实用的。

在警报和升级系统中实施静默时，警报仍然触发，但是不采取任何行动。这是一个重要的区别。警报仍然是触发的，我们只是不接收通知。问题或者停机仍然会发生，所有仪表盘或者显示设备仍然反映这一事实。

另外，可以在实时分析系统中实现静默。这种方法被称作抑制（Inhibit）。在这种情况下，警报不会触发，因此不发送任何警报。抑制可以通过一种机制实现，即一条警报规则规定在当前触发一个或者多个作为先决条件的警报规则时，某条警报规则应该不进行评估（计算）。另外，公式语言可以包含一个布尔函数，根据警报是否触发返回真或假。这个函数用于短路规则的评估。

例如，警告高错误率的一条警报规则可以在所用数据库离线时抑制。对于无法采取任

何行动的事情发出警报毫无意义。数据库监控由另一条规则（或者另一个团队）处理，这条警报规则可能是这样的：

```
IF http-500-rate > 1% THEN
  ALERT(error-rate-too-high)
  UNLESS ACTIVE_ALERT(database-offline)
```

静默条件创建 UI 建议

我们已经知道，人们不擅长进行与日期、时间和时区相关的算术运算，特别是在压力很大、警报不断的时候。在使用通配符或者正则表达式规定所要静默的警报时，也很容易犯错误。因此，我们建议采用如下 UI 功能：

- ☐ 默认开始时间应该是“现在”。
- ☐ 在结束时间上应该可以输入分钟、小时或者天数表示的时长，以及任何时区中的特定时间和日期。
- ☐ 用户可以检查自己的工作，UI 应该显示那些警报将被静默，在激活前需要确认。

抑制警报可能导致混乱，在运行中断发生时，由于抑制，监控系统会显示一切正常。在仪表盘显示正常运行，但服务却因为运行中断而失效时，依赖于服务的团队会感到困惑。因此，我们建议审慎使用抑制。

静默和抑制之间的差别很细微。当你根据某些条件（例如计划内运行中断或者升级）确定向某人发出传呼信号是错误时，可以使某条警报静默。抑制警报是为了在另一个条件成立时导致某条警报不触发。

17.5 可视化

可视化是创建一个或者多个指标的可视化表现形式，这不仅是创建取悦管理层的漂亮图表，还有助于找出大量数据中的意义。

设计优良的监控系统不会为了实时警报和可视化而两次收集收据。如果收集用于可视化的数据，就应该可以据此发出警报。如果收集用于警报的数据，保存这些数据就可以用于可视化。

简单的图标可以显示原始数据、摘要数据或者两个指标的对比。可视化还包括从其他指标合成新指标。例如，速率可以作为计数器和时间的函数计算。

一段时间的原始数据图表，不管是作为单数据点还是类似指标的聚合，对于缓慢变化的指标都很有用。例如，如果在一段时间内服务器内存的使用（通常几乎保持恒定）在特定软件发行之后的变化，就值得调查。如果使用率的上升出乎开发人员的意料，可能存在内存泄漏。如果使用率的下降出乎开发人员的意料，可能存在缺陷。

最好将计数器看成速率。以原始形式进行可视化，计数器看起来就像向右上延伸的直

线。如果计算一段时间的速率，我们将会看到加速、减速或者匀速。

有些可视化用处不大，实际上还会造成误导。已使用和未使用磁盘空间的饼图是常见的仪表盘显示，我们发现它很漂亮但是没有什么意义。如果不知道磁盘的大小，知道磁盘剩余 30% 的空间没什么用处。将两个不同大小磁盘的饼图放在一起观察，这种比较很少有用处。

相比之下，计算磁盘填充的速度并显示速度的导数，这一指标可以作为行动的依据。它有助于预测磁盘耗尽的时间。将这个导数绘制成图表，和其他资源消耗速度（如创建的新账户数）的导数对比，就可以看出不同资源消耗速度之间的比率。

唯一比饼图更不受我们欢迎的是平均数（算术平均值）。平均数可能误导你做出错误的决策。如果客户中一半喜欢热茶、一半喜欢冷茶，而你提供等于两种温度平均值的微温茶，那么所有客户都不会喜欢。在白天超载而在夜晚很少使用的网络链路平均使用率为 50%，根据平均利用率做出决策将是一个坏主意。平均值会撒谎。如果亿万富翁比尔·盖茨走进流浪汉收容所，这座建筑物里的人平均起来就是百万富翁了，但是这并不意味着无家可归的问题已经解决了。

话虽如此，平均数并不一定会造成误导，在你的统计工具箱中应有一席之地。和其他统计功能一样，必须明智地使用它们。

17.5.1 百分位数

百分位数和中位数在分析利用率或者质量时很有用。这些术语往往引起错误的理解。理解它们有一个简单的方法，想象一个学校有 400 名学生。让这些学生按照平均成绩点数（GPA）最低到最高的顺序排成 100 米长的一行，队列均匀分布，使平均成绩最低的学生站在 0 米处，最高的学生站在 100 米处。站在 90 米标志的学生是第 90 个百分位数。这意味着，90% 的学生成绩低于这名学生。在 50 米标志（正中）的学生被称作中位数。

如果学生更多或者更少，或者分级标准变化，总有一名学生在（或者最接近）90 米标志处。第 90 个百分位数上的学生的 GPA 可能是 C- 也可能是 A+。根据整个学校的 GPA 组成，整个学校可能都得到了出色的成绩，但是仍然有人处于第 90 个百分位。例如，可以用图表绘出每年第 50 和第 90 个百分位学生的 GPA，分析其趋势。

主机托管设施常根据使用带宽的第 90 个百分位数收费。这意味着它们每 5 分钟计量该时刻使用的带宽。所有这些计量进行排序，但是不删除重复的数值。如果你沿着 100 米长的足球场写下这些指标，分布均匀，则在 90 米标志的计量就是记账用的带宽率。这比按照最大值或者平均带宽收费更公平。

主机托管机构这样计费是非常聪明的。如果根据平均使用带宽收费，该公司将以每天传输的总字节数除以一天的秒数，并使用这一费率。这一费率对于夜间消费带宽很少，但是在白天消耗带宽很大的客户（称作昼行使用模式，diurnal）来说是有误导性的。相反，第 90 个百分位数代表了使用带宽的程度，而又不因为偶然的突发带宽消耗而惩罚你。实际上，

它放过了消耗带宽最高的 30 个 5 分钟周期，每天将近 2.5 个小时。更重要的是，对于主机托管机构，这最能体现使用带宽的成本。

在讨论单位延迟时也常使用百分位数。例如，一个网站试图改进页面加载时间，它可能设置一个标准——页面加载时间的第 80 个百分位数不高于 300 毫秒。特定 API 调用可能因为缓存、请求类型等而花费不同的时间，可以设置一个标准——第 99 个百分位数低于某个数值，如 100 毫秒。

17.5.2 堆栈排名

堆栈排名 (Stack ranking) 通过按值排序元素，简化了数据的对比。图 17.2 是堆栈排名 (Stack Ranking) 的一个例子。如果城市按照字母顺序排序，会不是很清晰，如亚特兰大和纽约比较。按照堆栈排名，我们可以清晰地看出差别。

堆栈排名最适合用于数据的同类比较。实现这种比较的方法之一是按照单位规范化数据 (例如，按照人口、机器、服务、每千次查询)。如果图 17.2 的 y 轴是城市里所有人吃掉的玉米饼总量，和规范化为人平均指标的 y 轴相比，图表的含义大不相同。后者是更容易对比的数字。

如果没有基准，堆栈排名可以帮助你确立一个。你可能不知道“55”是多还是少、不好还是好的数字，但是你知道只有两个城市的数字较大。不擅长确定雇员是否高效的经理可以依赖同行的堆栈排名而不是自己做出决策。

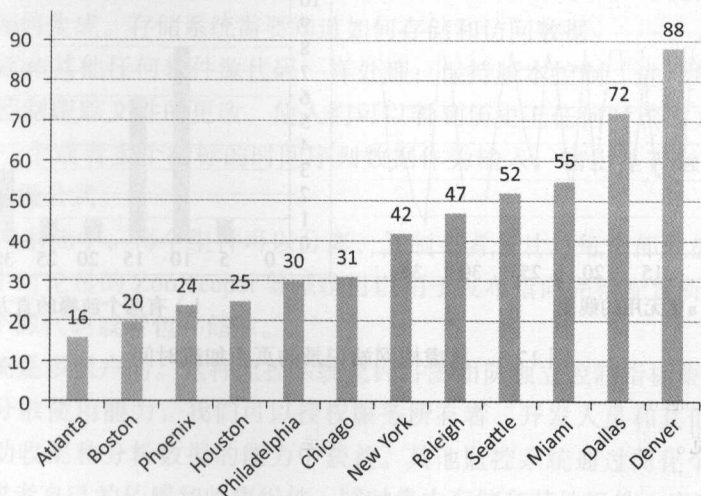


图 17.2 按照消费的玉米饼数量进行堆栈排序的城市

17.5.3 直方图

直方图可以揭示许多关于数据集的信息，在分析延迟时特别有用。直方图是数据集中数值计数的图表，这些计数可能经过舍入。例如，如果我们取一个数据集，并将所有数值

舍入为最近的 5 的倍数，就可以绘出表示 0、5、10、15 等数字个数的图表。我们可以简单地舍入为最近的整数。这些范围往往被称作“桶”（Bucket），每个数据点被放入最合适的桶中，这种图表可视化了每个桶中的项目数量。

图 17.3a 显示了一个虚构网站上收集到的页面加载计量。将这些计量绘制成线图除了表现出有许多不同的数字之外，没有揭示出太多其他信息。平均值是 27，也无法据此采取行动。

但是，如果我们将每个计量舍入为最近的 5 的倍数，并且绘制 0、5、10 等数字个数的图表，就可以得到图 17.3b 中的图表。这揭示了大部分数据点处于 10、15、35 和 40。这个图表有两个波峰，就像骆驼一样。这种模式称作双峰分布，知道数据有这种模式为我们提供了研究的基础。我们可以分离两个波峰内的数据点，搜索其相似之处。

我们可能发现，该网站的页面加载非常快，但是第一个波峰中的大部分数据点来自于特定的某个页面。调查显示每当生成这个页面，必须排序大量数据。重新排序数据，就可以缩短该页面的加载时间。另一个波峰可能来自需要某个数据库查找操作的网页，查找的数据大于缓存，因而性能不佳。我们可以调整缓存，改进该页面的性能。

可视化数据的其他方法还很多。更复杂的可视化可能提取不同的趋势，将更多信息合并为一个图表。可以使用颜色增加另一维，关于这个主题有许多很好的书籍，我们强烈推荐 Tufte 所著的《The Visual Display of Quantitative Information》（1986）。

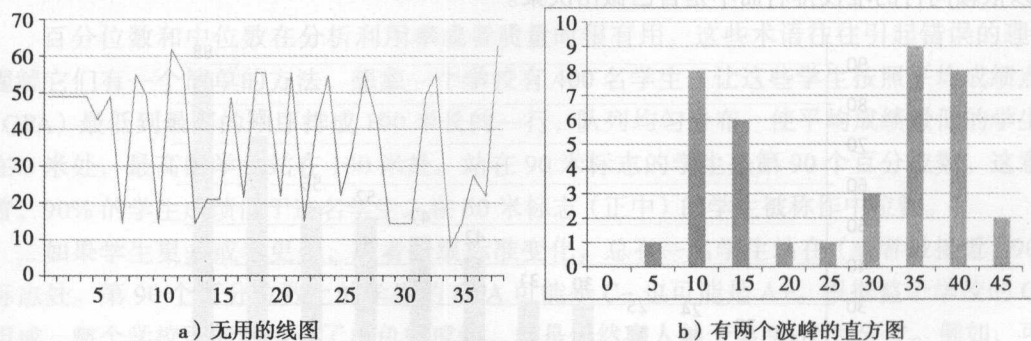


图 17.3 从虚构网站记录的页面加载时间

17.6 存储

存储系统保存收集的指标，使其可供其他模块访问。

存储是监控系统中架构要求最高的部件之一。新指标不断高速流入，警报需要快速、实时地读取最新数据。与此同时，其他分析需要大范围的迭代，使缓存变得很困难。典型的 SQL 数据库不擅长这些操作。

中等规模的系统往往对每个服务器收集 25~200 个指标。每台机器上有多个服务器。

中等规模的系统可能每秒需要保存 400 个新指标。更大的系统通常每秒存储数千个指标。全球分布的系统可能每秒保存数万到数十万个指标。

因此,许多存储系统处理实时数据或者长期存储,但是无法兼顾两者,或者无法在很大的规模上兼顾。最近,一些时间序列数据库(如 OpenTSDB)兴起,它们是专门为兼顾实时及长期存储所设计的,通过在 RAM 中保存最新数据(往往长达一个小时)并使用精心调整的存储格式实现上述目标。

时间序列数据的磁盘存储通常以两种方式之一进行。一种方法是通过固定大小的记录实现快速随机存取。例如,每个指标记录在 20 字节的记录中。系统可以使用改良的二分搜索法高效地找到特定时间的指标。这种方法在需要实时可视化时特别有效。另一种方法是压缩数据,利用增量可用很少的位数存储这一事实。结果往往是可变长度的记录,意味着时间序列数据必须从开始读取,找出特定时间的指标。这些系统能够得到更大的存储密度,有些系统在具备内建压缩功能的文件系统上存储固定大小记录,以达到两者的平衡。

17.7 配置

目前为止讨论的 6 个监控组件都需要配置信息指导工作。传感系统需要知道所要计量的数据和计量的频率。收集系统需要知道所收集的信息和发送的目标。分析系统需要处理公式库。警报系统需要知道向谁发出警报、如何发送以及升级到谁。可视化系统需要知道生成哪种图表,如何生成。存储系统需要知道如何存储和访问数据。

这些配置应该和其他任何软件源代码一样处理:保持版本控制、进行单元测试和系统测试等等。版本控制跟踪文件的更改,使人们可以看到历史上任何时点的文件内容。单元测试框架可以将一个或者多个指标的时间序列数据作为输入,输出是否触发警报的决策。这样就可以验证警报公式。

在分布式监控系统中,每个组件可以分离、复制或者分片。每个部分都需要访问配置的手段。11.7 节中讨论过的 ZooKeeper 等系统可以用于发布指向完整配置所在位置的指针,该位置往往是一个源代码或者包存储库。

有些监控系统是多租户的。这种监控系统允许许多团队独立控制指标集、警报规则等。通过集中服务、分散使用能力,我们可以授权服务所有者、开发人员和其他人进行自己的监控,同时从自动收集和分析数据的能力中获益。其他监控系统通过简化个人安装自己的监控系统实例,或者自己的传感和收集组件,同时集中存储和其他组件,实现相同目标。

17.8 小结

监控系统很复杂,许多组件共同工作。传感和计量组件获取计量指标。白盒监控系统内部。黑盒从用户视角收集数据。仪表计量变化的数量。计数器是某些事件发生次数的不

递减指示。收集系统收集指标，指标可以推送（发送）到收集系统，或者由收集系统向监控系统拉取（查询）。

存储系统保存指标，通常使用定制的数据库处理大量输入数据，并利用时间序列数据的独特性质。

分析系统从数据中提取含义。可能有许多不同的分析系统，每个系统提供异常检测、预测或者数据挖掘等服务。有些分析实时进行，在收集数据时发生。短期分析专注于最新数据，或者提供特定应用程序（如仪表盘）所需的随机访问。长期分析检查大跨度数据，发现许多年间的趋势。这种分析往往以批量模式进行，存储中间结果供以后使用。

警报和升级系统在需要人工干预时寻找合适的人员，当这个人在固定时间内没有响应时寻找替代人员。

可视化系统提供图表和仪表盘。它们可以合并和转换数据，并进行计算百分位数、构建直方图和确定堆栈排名等操作。

所有这一切都由配置管理器联系起来，指导其他组件的工作。对配置的更改可以许多方式分发，常用的方式是分发配置文件或者 ZooKeeper 等更动态的系统。

监控系统是多租户时，我们授权单独的服务团队控制自己的监控配置。他们从集中化组件中获益，不需要担心容量规划和其他运营任务。

所有组件一起工作时，我们就可以得到可伸缩、可靠和有效运转的监控系统。

练习

1. 监控系统有哪些组件？
2. 选出 3 个监控系统组件并详加描述。
3. 所有监控系统都有本章描述的所有组件吗？举例说明组件可以选择的原因。
4. 什么是“传呼风暴”？如何处理？
5. 研究表示数据的 JSON 格式。为指标收集设计一个 JSON 格式。
6. 描述你的组织中使用或者你过去遇见的监控系统。它们如何使用？监控哪些内容？解决什么问题？
7. 创建计算计数器指标比率的一种或者多种方法。该方法应该在计数器重置时仍然正常工作。你的方法是否能计算误差范围？
8. 为当前环境设计更好的监控系统。
9. 为什么不鼓励使用平均数？举出一个平均数会产生误导，但是其他方式不产生误导的指标例子。

容量规划

计划本身并不重要，计划的过程重于一切。

——Philip Kotler

容量规划意味着确保在需要时有足够的资源。最优的做法是使系统既不会容量不足，也不会过多。资源包括 CPU、内存、存储设备、服务器实例、网络带宽、交换机端口、控制台连接、电源、散热、数据中心空间和任何其他运行服务必需的基础设施组件。

容量规划有两个主要目标。首先，我们想要避免因为容量缺乏而导致服务中断。其次，我们想要在任何时刻只添加必要的容量，节约资金投入。好的容量规划说明资源需求，将资源使用量控制在确保优良服务的限度下，提供明显的投资回报（Return On Investment, ROI）。容量规划应该是数据驱动的过程，使用收集到的运行系统有关信息预测趋势，它还应该遵循未来的业务成长计划。本章解释需要收集的数据，以及用这些信息预测未来容量需求的方法。

在快速成长的大型服务中，容量规划很复杂，依赖于精密的数学模型。组织往往雇用全职的统计人员开发和维护这些模型。具有技术背景的统计人员可能很难找到，但是这种人员值得高薪聘用。本章介绍为金融市场交易开发的一些数学模型，并说明如何将它们应用到快速变化环境的容量规划中。

本章关注的不是满足为服务提供商工作的人员日常需求的企业风格的容量规划，而是将重点放在服务本身的容量规划上。

必知术语

QPS：每秒查询数，通常是每秒接收的 Web 点击数量或者 API 调用数量。

活动用户数：在特定时限内访问服务的用户数。

MAU：每月活跃用户。在上一个月中访问过服务的用户数。

参与度：活动用户执行特定事务的平均次数。

主要资源：作为服务主要限制因素的一个系统级资源。

容量限制：性能开始快速降级或者变得无法预测的点。

核心驱动力：强力推动主要资源需求的因素。

时间序列：在相同时空间隔测量的一系列数据点。例如，来自监控系统的数据。

18.1 标准容量规划

容量规划需要提供两个问题的答案：你需要在下一年中购买什么？何时需要购买？为了解答这些问题，你需要知道如下信息：

- ❑ **当前使用情况：**哪些组件可能影响服务能力？现在每个组件的使用量是多大？
- ❑ **正常增长：**在没有特定业务或者市场事件影响的情况下，预期的服务增长率是多少？有时候这被称作**自然增长**（Organic Growth）。
- ❑ **计划增长：**计划了哪些业务或者市场活动，这些活动何时发生，由于这些活动预计产生多大的增长？
- ❑ **余量：**你的服务遇到何种短期使用高峰？明年是否有特定的事件，如奥运会或者选举，可能导致使用高峰？需要多少备用容量才能“优雅”地处理这些高峰？余量通常以当前容量的百分比指定。
- ❑ **时间表：**对于每个组件，从订购到交付以及从交付到投入服务之间的提前期是多长？将新容量引入服务有没有特定的约束，如更换窗口？

数学术语

相关系数：描述不同数据源计量相互之间的相似程度。

移动平均数：一系列平均数，每个都取自于一个较短的时间间隔（窗口），而不是取自整个数据集。

回归分析：一种分析不同数据源之间关系，确定其相关性并根据一个数据源的变化预测另一个数据源变化的统计学方法。

EMA：指数移动平均数。对窗口中的每个数据点应用一个权重，旧数据点的权重以指数方式递减。

MACD：移动异同平均数。这一指标用于发现指标强度、方向和动力中的变化。它测量短窗口 EMA 和长窗口 EMA 之间的差别。

零线交叉：MACD 线交叉通过零点发生在短 EMA 和长 EMA 之间没有差异的时候。

由正转负代表数据的向下趋势，由负转正表示数据的向上趋势。

MACD 信号线：MACD 计量的 EMA。

信号线交叉：MACD 线与信号线交叉代表数据中的趋势将向交叉的方向加速。这是一个动力指标。

根据上述信息，可以用一个简单的公式计算下一年底每种资源的预期容量需求：

$$\text{未来资源} = \text{当前使用量} \times (1 + \text{正常增长} + \text{计划增长}) + \text{余量}$$

你可以计算需要购买的每种资源附加容量：

$$\text{附加资源} = \text{未来资源} - \text{当前资源}$$

为每个资源执行这一计算，不管你是否认为需要更多容量。得出来年不需要更多网络带宽的结论也是正常的。但是因为没有在容量规划中考虑，而意外地耗尽网络带宽则是不行的。对于共享资源，需要结合来自许多团队的数据以确定是否需要更多容量。

18.1.1 当前使用量

在你考虑购买附加设备时，需要理解当前可用的设备以及使用的数量。在评估你所拥有的设备之前，需要提供服务所需的一切设备的完整列表。如果忘记了某个项目，它就不会包含在容量规划中，你可能在以后耗尽某项资源，结果无法按照需要快速扩张服务。

需要跟踪的资源

基于互联网的服务提供者最明显需要的两种资源是提供服务的机器和互联网连接。有些机器可能是在以后为实现指定任务定制的通用机器，其他机器则可能是专用设备。深入这些项目，机器有 CPU、缓存、RAM、存储器和网络。连接到互联网需要局域网、路由器、交换机和至少一个 ISP 的连接。再深入一步，网卡、路由器、交换机、电缆和存储设备都有带宽的限制。有些设施可能具有需要特殊接线和网络设备接口的高端网卡。所有联网设备都需要 IP 地址，这些都是需要跟踪的资源。

退后一步，所有设备都运行某种操作系统，有些还运行附加软件。操作系统和软件可能需要许可证和维护合同。设备的数据和配置信息需要备份到更多的系统。再退一步，机器必须安装在符合电源及环境需求的数据中心。数据中心的机架数量和类型，电源和散热容量以及可用的占地面积都需要跟踪。数据中心可能提供附加的按机器服务，如控制台服务。对于有多个数据中心和接入点的公司，这些站点之间的链路可能也有容量限制，这些都是需要跟踪的附加资源。

外部供应商可能提供某些服务。涵盖这些服务的合同规定成本或者容量限制。为了覆盖所有可能的方面，应该和每个部门的人交谈，找出他们所做的工作以及和服务的关联。对于和服务关联的任何因素，都应该理解其限制、跟踪方法以及计量可用容量的方法。

你拥有多少

要跟踪资产，保持最新库存数据库是不可替代的方法。库存数据库应该作为订购、配给和退役过程中的核心组件，保持最新状态。保持最新的库存系统为你提供找出拥有的各项资源所需的数据。它还应该用于跟踪软件许可证和维护合同库存，以及可从第三方取得的协议资源量。

使用有限的标准机器配置和一组标准设施、存储系统、路由器和交换机，就可以更轻松地设备数量映射到较低级别的资源，如它们所提供的 CPU 和 RAM。

现在使用了多少

确定每个服务的限制性资源。你的监控系统可能已经收集了 CPU、RAM、存储和带宽的资源使用数据。通常它收集数据的频率高于容量规划的需求。摘要或统计样本对资源规划已经足够，而且通常可以简化计算。将这些数据与来自库存系统的数据结合，就能说明当前拥有的备用容量。

跟踪库存数据库中的所有项目并使用一组有限的标准硬件配置，还可以轻松地指定每个设备使用的空间、电源、散热和其他数据中心资源。将这些数据全部输入库存系统，就可以自动生成数据中心利用率。

18.1.2 正常增长

监控系统直接提供关于当前使用量和当前容量的数据，它还可以提供前几年的正常增长率。寻找使用率的显著跳跃性变化，看看它们是否对应于特定事件，如新产品试运行或者特殊市场推动活动。如果因为该事件在这一年的余下时间持续而造成偏差，计算变化并将其从后续数据中减除，避免在正常增长率计算中包含这种事件驱动的变化。在图表上绘制尽可能多年份的数据，确定正常增长率是否呈线性或者遵循其他趋势。

18.1.3 计划增长

第二步是估算由于市场和业务事件（如新产品投放或者新功能）造成的额外增长。例如，市场部门可能计划在 5 月份发起一项重大活动，预计将增加 20%~25% 的客户。或者 8 月份计划启动依赖于 3 个现有服务的新产品，预计在投放时给每个服务增加 10% 的负载，到年终增加到 30%。使用第一步检测到的任何变化数据，验证预期增长的假设。

18.1.4 余量

余量是被视为正常的过剩容量。任何服务都有使用高峰或者边缘条件，需要偶然扩大资源使用量。为了避免这种边缘条件触发运行中断，在日常工作中必须有备用资源。任何给定服务所需的余量都是业务决策。由于过剩容量在大部分情况下不使用，当然就代表着可能浪费的投资。因此，在财务上负责任的公司希望平衡服务中断的可能性和节约财务资

源的期望。

监控数据应该选择这些资源高峰并提供它们何时、何地 and 如何发生的过硬统计数据。运行中断和事后剖析报告数据也是决定合理余量的关键。

确定所需余量的另一个因素是从发现需要附加资源开始到附加资源投产之间花费的时间。如果需要 3 个月才能使新资源可用，就需要比 2 周或者 1 个月就能得到新资源的情况下更多的余量。至少，你需要足够的余量，以便应付这段时期的预期增长。

18.1.5 弹性

可靠的服务还需要附加容量以满足其 SLA。在有些组件发生故障时，这些附加容量可以保证最终用户不会遭遇停机或者服务降级。正如第 6 章中所讨论的，附加容量必须处于不同的故障域。否则，一次运行中断就会造成主机和应该接管负载的备用容量都下线。

故障域还应该在更大的规模上考虑，通常是数据中心的级别。例如，对电源系统进行的设备级维护需要关闭整个建筑物的电源。如果整个数据中心都下线，服务必须从其他数据中心顺畅运行而没有容量问题。将服务容量分布到许多故障域降低了处理弹性需求所需的额外容量，这是提供这种额外容量最具成本效益的方法。例如，如果某个服务在一个数据中心运行，第二个数据中心必须提供附加的容量（大约 50%）。如果一个服务运行于 9 个数据中心，需要第 10 个数据中心提供附加容量，这一配置只需要 10% 的附加容量。

正如 6.6.5 节中所讨论的，“黄金标准”是提供足以应付两个数据中心同时停运的足够容量。这可以在一个数据中心因为计划内维护而停运的同时，组织为另一个可能意外停运的数据中心做好准备。附录 B 讨论这种弹性架构的历史。

18.1.6 时间表

大部分公司每年计划其预算，并将开支划分到各个季度。根据预计的正常增长和计划内的突发增长，可以描绘出何时需要投入资源。从这一天倒推，需要知道资源可用所需的时长。购买订单得到批准、发给供应商需要多长时间？从接受购买订单到供应商交付产品需要多长时间？从交付到资源可用需要多少时间？在设备安装之前是否需要执行特殊测试？开启额外容量是否需要特定的更改窗口？一旦开启附加容量，重新配置服务利用这些容量需要多长时间？利用这些信息，可以提供开支时间表。

物理服务的提前期通常长于虚拟服务。IaaS 和 PaaS 服务（如 Amazon 的 EC2 和 Elastic Storage）流行的部分原因是新请求的资源实际上是立即交付的。

缩短资源交付时间总是有很高的成本效益，因为这意味着我们为覆盖资源交付时间所付出的过剩容量更少。这是使用准备新获得资源的自动化系统有直接价值的地方。

18.2 高级容量规划

高速增长的大型环境（如流行的互联网服务）需要不同的容量规划方法。标准企业风格容量规划技术往往是不够的。客户基础可能快速变化、难以预测，需要更深入和更频繁地对服务监控数据进行统计分析，更快地发现使用量趋势中的显著变化。这种容量规划需要更深厚的技术知识。容量规划人员需要熟悉 QPS、活跃用户、参与度、主资源、容量限制和核心驱动力等概念。Yan 和 Kejariwal（2013）介绍过本节描述的技术，他们的工作给本节的内容带来了启发。

18.2.1 确定主要资源

每个服务都有一个**主要资源**（Primary Resource），如 CPU 利用率、内存占用或者带宽、存储占用或带宽、网络带宽等，是服务消耗资源中占据主导地位的部分。例如，进行大量计算的服务往往受限于可用的 CPU 资源——它是 CPU 受限的服务。容量规划的重点就是主要资源。

服务还有**次要资源**（Secondary Resource）的需求。例如，CPU 受限服务可能在较小程度上使用内存、存储和网络带宽。在现行软件版本和硬件型号下，次要资源从容量规划的角度看不受重视，但是它们中的某种资源可能由于代码或者硬件的变化而成为主要资源。因此，这些次要资源也应该受到监控，跟踪其使用趋势。发现指定服务主要资源的变化主要通过跟踪主资源和次要资源之间的约束比率（磁盘：CPU：内存：网络）来完成。每当新软件版本发行或者新硬件型号推出，应该重新计算这一比率。

主要和次要资源是低级资源单位，驱动**辅助资源**（Ancillary Resource）（如服务器实例、交换机端口、负载均衡器、电源和其他数据中心基础设施）的需求。

容量规划的第一步是确定主要资源，因为它们是容量规划中关注的资源。你还必须定义主要和辅助资源使用量之间的关系。当容量规划表明需要更多主要资源时，必须能够确定所需的辅助资源的数量。当使用的硬件变化，这些对应关系也必须更新。

18.2.2 了解容量限制

任何资源的**容量限制**（Capacity Limit）是性能开始快速降级或者变得无法预测的点，如图 18.1 所示。容量限制应该通过负载测试确定。负载测试通常在隔离的实验室环境进行，可以使用合成的流量或者重放生产流量进行。

对于每个主要和次要资源，都必须知道其容量限制。为了独立生成每个资源的数据，你的实验室设置应该为除了一个低级资源之外的所有资源配备大量过剩容量。然后，该资源成为服务的限制资源，根据响应时间和该资源的利用率及资源可用绝对数量的对比单独绘制图表。为了完整性，最好多次重复测试，每次都统一地按比例扩展整个环境。这种方法可以确定限制是和资源利用率还是剩余资源数量更紧密相关。

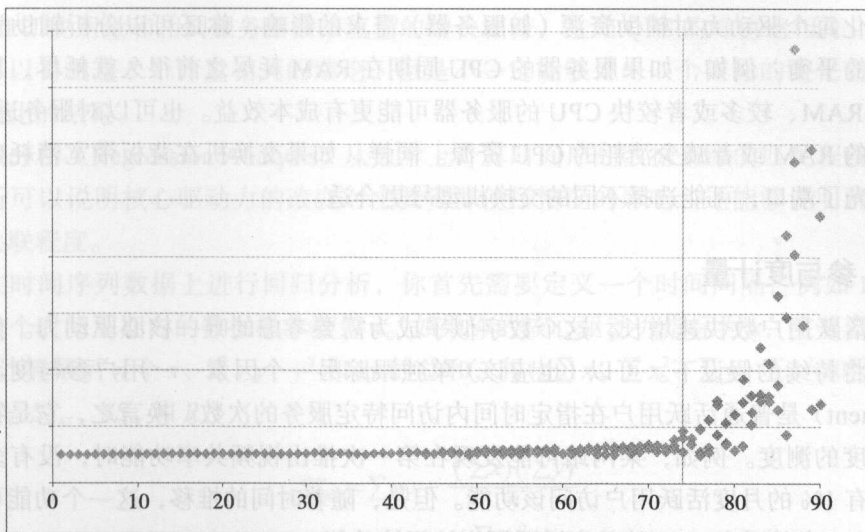


图 18.1 超出某个利用率之后，响应时间开始降级

18.2.3 确定核心驱动力

核心驱动力 (Core Driver) 是强烈驱动主要资源需求的因素。它们通常包括 MAU、QPS、语料库大小或者其他高级指标值，这些指标代表了在服务上造成流量或者负载的因素。这些指标往往有业务上的意义，例如和收入来源有联系。

网站可能有数百万注册用户，但是活跃的用户通常少得多。例如，许多人注册账户并使用服务一两次，但是再没有回来。在规划中考虑这些用户可能产生误导。许多人在社会化网络网站上注册，但是很少使用账户。在购物网站注册的有些人可能只在生日之前和需要购买礼物的节日之前使用服务。用户数更精确的代表是最近 7 天或者 30 天中活跃的用户数量。最近 7 天中的用户数往往称作 **7 日活跃用户 (7-Day Active, 7DA)**，而**每周活跃用户 (Weekly Active User, WAU)** 指的是特定日历周中活跃的用户数。同样，**30 日活跃用户数 (30-Day Active, 30DA)** 计量最近 30 天的用户数，而**月度活跃用户 (Monthly Active User, MAU)** 用于特定日历月的计量。这些计量往往比注册用户数更准确地反映使用量。

对于具有时间成分的活跃用户等指标，在容量规划中，不同服务适合使用不同的时间成分值。例如，月度活跃用户适合作为某些服务的核心驱动力，而对于另一些服务，容量规划中使用每分钟活跃用户可能更合适。对于活动用户驱动的高事务性系统，较小的时间尺度（如每分钟活跃用户数）可能比较合适。对于用户驱动的存储受限服务，注册用户总数（或者总用户语料库数）可能更合适。

容量模型 (Capacity Model) 描述了核心驱动力和主要资源之间的关系。对于指定的服务，容量模型表示核心驱动力的变化如何影响服务的需求。

一旦确定核心驱动力，并且确定每个核心驱动力对每个主要资源和次要资源的影响，

就可以量化每个驱动力对辅助资源（如服务器）需求的影响。你还可以分析辅助资源是否达到很好的平衡。例如，如果服务器的 CPU 周期在 RAM 耗尽之前很久就耗尽，那么购买具有较少 RAM、较多或者较快 CPU 的服务器可能更有成本效益。也可以对服务进行返工，利用额外的 RAM 或者减少消耗的 CPU 资源。同样，如果交换机在背板带宽消耗殆尽之前很久就用光了端口，可能选择不同的交换机型号更合适。

18.2.4 参与度计量

如果活跃用户数快速增长，这个数字似乎成为需要考虑的唯一核心驱动力。但是，这是在增长将持续的假设下。可以（也应该）单独跟踪另一个因素——用户参与度。**参与度（Engagement）**是普通活跃用户在指定时间内访问特定服务的次数。换言之，它是特定功能受欢迎程度的测度。例如，某网站可能发现在第一次推出视频共享功能时，没有多少人感兴趣，只有 1% 的月度活跃用户访问该功能。但是，随着时间的推移，这一个功能可能变得更加流行，6 个月后有 25% 的月度活跃用户访问该功能。

如果活跃用户数量保持不变，但是参与度提高，则服务的负载也将增大。结合这两个数字，可以说明服务被访问的频率。

每个服务的参与度图表都不同，一个服务的参与度提高，对资源需求的影响与另一个服务的参与度变化对资源需求的影响会有不同。例如，视频上传参与度的提高对磁盘空间和 I/O 带宽的影响要大于聊天服务参与度提高带来的影响。

18.2.5 分析数据

一旦决定了所有收集的指标并开始收集数据，就需要处理数据，为容量规划过程生成有用的输入。标准容量规划关注同比统计数字，根据这些数据做出预测。这种方法仍然是必要的，但是在快速变化的大规模环境，我们需要扩充该方法，更快地对需求的变化做出响应。

在大规模环境中，目标之一是可以简单而准确地根据测得的核心驱动力决定所需的资源数量。实现这一目标的方法之一是根据一个机架或者机器群集的服务容量将指标简化为用户分块。利用这种方法，容量规划被简化为千篇一律的方法。某个应用程序可能在每个机架上服务 10 万活跃用户。另一个应用程序可能在每个群集中服务 1000 个活跃用户，每个群集是作为独立单元的标准化机器组合。在更快的硬件出现或者创建新软件设计时，工程人员可以生成新的比率。现在容量规划得到简化，资源可以根据活跃用户块数管理。这种方法虽然较不细致，但是已经足够，因为它和部署的粒度相符。你不能购买半台机器，所以不需要特别精确的容量规划。

为了确定核心驱动力和资源消耗之间的关系，首先必须理解哪些核心驱动力影响哪些资源，以及影响的强度。具体的做法是将资源使用指标和核心驱动力指标关联起来。

关联

关联（Correlation）计量数据源之间的相似度。从视觉上，你可能在监控图表上发现到

服务器 CPU 使用量和相同服务器网络流量的增长相符, 这和 QPS 的峰值也相符。从这些观察中, 可以得出这 3 个计量相关的结论, 但是不一定能够得出一个指标的变化会导致另一个指标变化的结论。

回归分析 (Regression Analysis) 从数学上计算时间序列数据来源的匹配程度。指标的回归分析可以说明核心驱动力的改变对主要资源使用情况的影响, 还能够说明两个核心竞争力的关联程度。

要在时间序列数据上进行回归分析, 你首先需要定义一个时间间隔, 例如 1 天或者 4 周。在这个时间周期内的数据样本数为 n 。如果你的核心驱动力指标为 x , 主要资源指标为 y , 首先计算最后 n 个 x 、 x^2 、 y 、 y^2 和 xy 的总和, 得出 $\sum x$ 、 $\sum x^2$ 、 $\sum y$ 、 $\sum y^2$ 和 $\sum xy$, 然后计算 SS_{xy} 、 SS_{xx} 、 SS_{yy} 和 R :

$$SS_{xy} = \sum xy - \frac{(\sum x)(\sum y)}{n}$$

$$SS_{xx} = \sum x^2 - \frac{(\sum x)^2}{n}$$

$$SS_{yy} = \sum y^2 - \frac{(\sum y)^2}{n}$$

$$r = \frac{SS_{xy}}{SS_{xx}SS_{yy}}$$

回归分析得到相关系数 R , 这是 -1 和 1 之间的数字。求这个数的平方再乘以 100 就可以得到两个数据源匹配的百分数。例如, 对于图 18.2 中所示的 MAU 和网络利用率数字, 这一计算可以得出非常高的相关 (96% 和 100% 之间), 如图 18.3 所示, 图中绘制了 R^2 的图表。

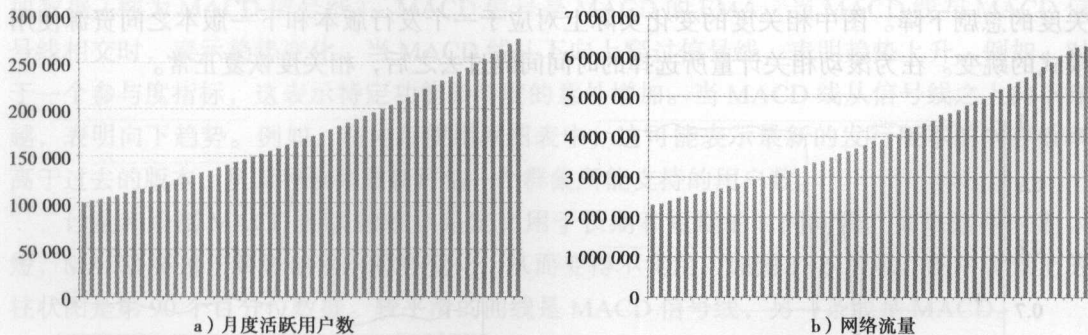


图 18.2 用户数和网络流量的相关度很高

当一个核心驱动力中的变化与某个主要资源使用率的变化有强相关时, 可以得出 $y = a + bx$ 形式的方程, 描述两者之间的关系, 这称作回归线 (Regression Line)。这一方程式使你可以根据核心驱动力计量计算主要资源需求。换言之, 给定核心驱动力 x 的值, 就可以

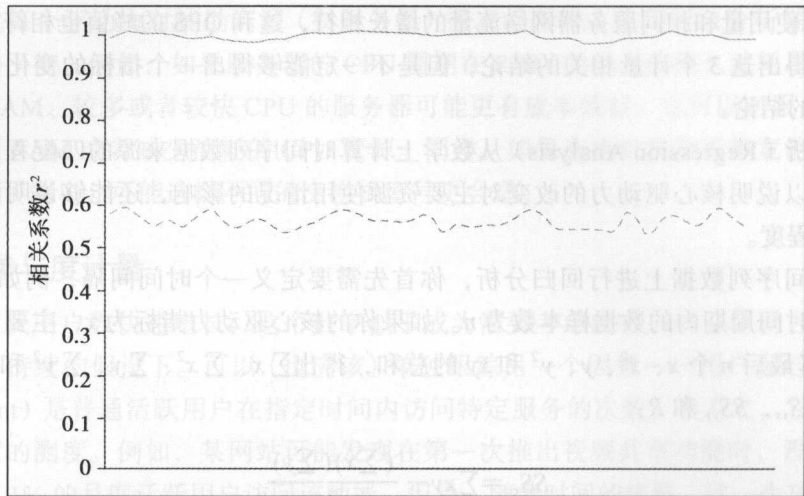


图 18.3 上方的曲线显示两个数据源之间有 96% 的高相关度。下方的曲线显示低相关度——不到 60%

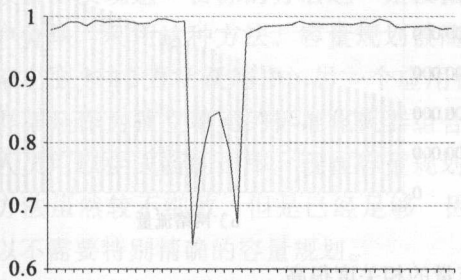
计算需要多少主要资源 y ，置信度为 R^2 。为了计算 a 和 b ，首先计算最后 n 个数据点 x 和 y 的移动平均数， \bar{x} 和 \bar{y} 。然后：

$$b = \frac{SS_{xx}}{SS_{yy}}$$

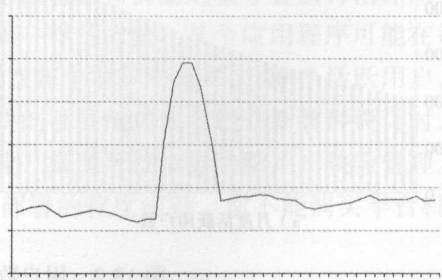
$$a = \bar{y} - b\bar{x}$$

指标之间的相关度随时间推移而变化，因此应该绘制图表，用滚动相关分析跟踪，而不是评估一次之后就假定其恒定。服务中的变化对相关有显著影响。最终用户人口统计特征的变化通常较慢，但是也会改变普通客户使用服务的方式，从而影响相关度。

图 18.4a 中显示，与一个软件发行版本和下一个版本之间资源使用模式的变化对应，相关度的急剧下降。图中相关度的变化实际上对应于一个发行版本和下一版本之间资源使用模式的跳变。在为滚动相关计量所选择的时间间隔过去之后，相关度恢复正常。



a) 相关度的突然变化



b) 乘数 b 的变化

图 18.4 MAU 和带宽之间相关度的变化

图 18.4b 显示了相同时间间隔中的 b 。注意，在升级之后， b 在关联分析所选择的时期

内显著变化, 然后再次稳定, 但是保持在较高的数值上。 b 在相关窗口长度中的大幅波动是因为每天移动平均数的显著变化, 这是由于移动平均数兼具升级前后的数据。经过充足的时间, 移动平均数中只使用升级后的数据, b 就恢复稳定, 相关系数也恢复到之前的高水平。

b 值对应于直线的斜率, 也就是联系核心驱动力及主要资源用量的方程式中的乘数。当相关度恢复正常时, b 处于较高的水平。这一结果表明, 主要资源在这一软件发行版本中的消耗快于前一个版本。相关度中的任何明显变化应该触发对乘数 b 及对应资源使用预测的重新评估。

预报

预报试图根据当前和过去的计量预测未来需求。最基本的预报技术是绘出历史使用量的第 90 个百分位, 然后找出最适合于此数据的公式。然后, 可以使用这个公式预测未来的使用量。17.5.1 节讨论百分位数的计算。

当然, 增长率变化、使用模式变化和应用程序资源需求也会变化。我们必须检测这些变化, 相应修改资源计划。为了检测趋势中的变化, 我们使用移动异同平均数 (MACD) 指标。MACD 计量长期 (例如 3 个月) 和短期 (例如 1 个月) 移动平均数之间的差异。但是, 标准移动平均数倾向于掩盖指标中最近的变化。由于预报的目标是尽早发现这些变化, MACD 使用指数移动平均数 (EMA), 为最近的数据提供更大的权重, 而对旧数据应用很低的权重, 计算平均值。EMA 的计算如下所示, 其中 n 为样本数量:

$$k = \frac{2}{n+1}$$

$$EMA_x = Value_x \times k + EMA_{(x-1)} \times (1-k)$$

首先, 第一个 EMA_{x-1} 值 (实际上是 EMA_n) 就是前 n 个数据点的直接平均值。

为了使用 MACD 提供行为变化的早期预警, 必须计算并在同一个图表中绘出一些附加数据 (称为 MACD 信号线)。MACD 信号是 MACD 的 EMA。当 MACD 线与 MACD 信号线相交时, 表示趋势变化。当 MACD 线从下向上穿过信号线, 表明趋势上升。例如, 对于一个参与度指标, 这表示特定功能参与度的意外增加。当 MACD 线从信号线之上向下穿越, 表明向下趋势。例如, 在内存使用率图表中, 这可能表示最新的发行版本的内存效率高于过去的版本, 从而导致你重新评估一个群集所能支持的用户数。

计量和绘制 MACD 图表的难点是定义用于长期和短期的时间间隔。如果这些时期太短, MACD 将过于频繁地表示趋势变化, 从而变得不实用, 如图 18.5 所示。图中背景上的柱状图是第 90 个百分位数据。较平滑的曲线是 MACD 信号线, 另一条线是 MACD。

但是, 增大短周期特别容易延迟趋势事件变化的触发。在图 18.6 中, 使用 2 周的短周期 (图 18.6a), 触发向下趋势的时间比使用 4 周的短周期 (图 18.6b) 时早两天 (长周期保持为 3 个月)。但是, 2 周的短周期还有少数额外的噪声, 在向下趋势触发 2 天之后又触发一个向上趋势, 一周之后最终触发向下趋势。

在选择长短周期时, 可以查看旧数据是否能预测较新的数据, 验证该模型。如果可以

预测，得出该模型能够很好地预测短期未来情况的结论就是合理的。从现有数据开始，一直倒推到你所拥有的所有数据。尝试不同的长短周期组合，观察哪个组合能够最好地预测出历史数据集中观察到的趋势。

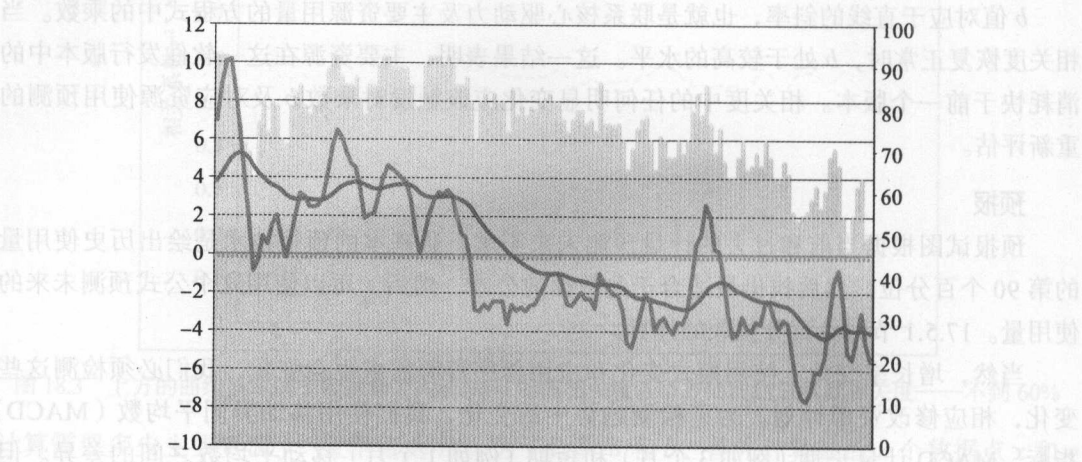
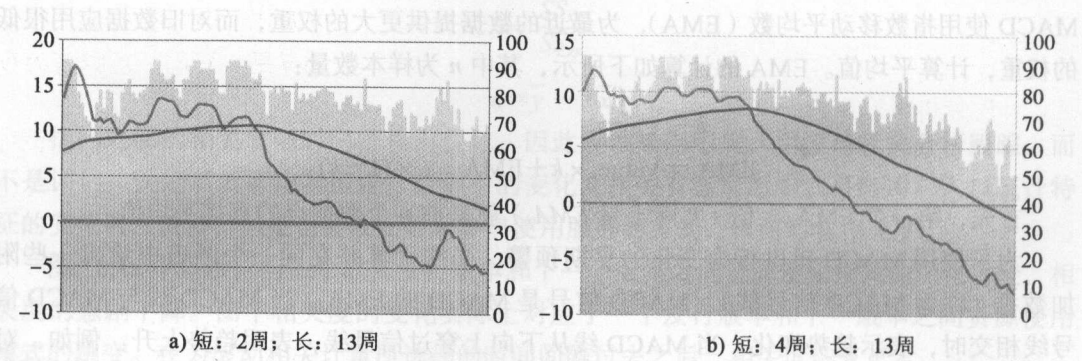


图 18.5 1 周短周期和 4 周长周期中的数据包含噪声



a) 短：2周；长：13周

b) 短：4周；长：13周

图 18.6 只改变短周期的影响

18.2.6 监控关键指标

核心驱动力和资源之间的相关系数应该绘制成图表加以监控。如果相关度有显著变化，核心驱动力和资源消耗之间的关系应该重新评估，并将任何变化反馈到容量规划过程中。

类似地，每个核心驱动力和资源的 MACD 和 MACD 信号也应该监控。由于这些指标可以用于及早发现趋势变化，它们是容量管理中的关键要素。MACD 线和 MACD 信号线交叉产生的警报应该直接发送给负责容量规划的团队。如果核心驱动力以及和主要资源的关系有很好的定义，理论上只需要监控核心驱动力和相关系数。实际上，监控所有指标以最大限度地降低出现意外的概率，是审慎的做法。

18.2.7 委派容量规划

容量规划往往由技术人员完成。但是，有出色的指标和核心驱动力对资源需求影响的清晰理解，就可以将容量规划与部署解耦。项目经理可以进行容量规划和订购，而技术人员则负责部署。

构建容量规划仪表盘，将其作为监控系统的一部分，这样，非技术人员也能够进行容量规划。创建一个或者多个以专用视图显示容量数据的网页，理想的情况下应该具有自动创建图表的能力。使这个仪表盘在组织中可以独立于主监控仪表盘访问。这样，组织中的任何一个人都是可以合理的方式访问数据，为额外容量的资本支出提供依据。

有时候，做出的决策是不购买更多资源而是更有效地利用现有资源。例子包括压缩数据而不购买更多存储设备和带宽，或者使用更好的算法而不增加内存或者 CPU。这些方案的测试依赖于资源回归，我们将在下面介绍。

18.3 资源回归

资源回归是计算一个发行版本和另一个发行版本之间资源使用量的差别。每个软件发行版本的资源需求都应该略有不同。如果新软件发行版本使用的资源显著增加，这种差异往往是计划之外的。换言之，这是应该报告的缺陷。如果这种差异是有意的，意味着容量规划人员必须调整模型和购买计划。

为了执行资源回归，根据与发行关联的用户事务进行 workflow 分析。这只是制作新发行版本中可能的功能清单以及每项功能资源消耗量的有趣说法而已。然后，对于每项功能，将每个客户的预计事务数乘以活跃客户数量。这将给出新发行版本的资源预测。

例如，假定你有一个照片共享网站，网站客户有 3 种主要事务。首先，用户可以登录和编辑自己的档案数据。其次，用户可以上传照片。最后，用户可以查看照片。每个事务都有可以计量的资源成本。如何计量？最简单的方法是分析来自预演系统和在你控制之下的样板用户的数据。

使用自动化测试脚本模拟用户登录和事务，可以拥有已知数量的用户，每个用户参与特定的事务。从该系统收集的监控数据可以和来自基线（没有模拟用户）系统的数据比较。可以合理地假定，内存、磁盘、CPU 和网络使用量的差异是因为用户事务产生的。从带负载的占用中减去基线系统占用的资源，然后将差值除以用户事务数量，就可以得到每个事务的资源使用量。一定要对预测进行负载测试，看看资源使用量是线性伸缩，还是在增加更多事务时出现拐点。

计算 workflow 分析中，一定要包含基础设施资源成本。对这个端到端的事务需要多少次 DNS 查找？多少次数据库调用？单一事务可能使用系统基础设施中的许多个服务，这些服务资源也必须被评估并随着事务服务器的伸缩而合理的伸缩。

18.4 发布新服务

现在，我们已经理解了现有系统的容量规划方法，下面谈谈新服务的发布。发布新服务是困难而危险的，因为没有事先的经验或者服务指标以规划所需的初始容量。对于大型服务，测试可能不可靠，因为预演环境可能没有足够的容量进行真正的测试。

更有甚者，发布时服务往往处于媒体最严密的监视之下。如果系统耗光容量而变得不可靠，媒体除了描述投放的糟糕状况之外别无选择，这会给客户留下不好的第一印象。

缓解这种风险的方法之一是找到慢慢增加活跃用户数量的途径。例如，启用产品但是不宣布（软启动）可以为工程师留出时间，找出和解决问题。但是，如果需要附加资源且需要几周的提前期，就只能采用最慢的用户数量升级。

依赖于缓慢升级用户数量对著名公司来说不可选择。Google 和 Facebook 多年以来没有在新服务上采用软发布方法慢慢升级用户数量。任何新服务都立即让新用户涌入。因此，对新服务进行准确的容量规划，已经成为高度渴求的技能。

幸运的是，可以用一种技术来弥补知识的不足，这种技术就是摸黑启动（Dark launch）。Facebook 在其聊天消息系统中使用了摸黑启动，确保该公司向其用户基础提供可靠的系统。

在摸黑启动中，新功能以模拟的流量发布到生产环境中，实际上将生产环境视为受到严格控制的新功能资源需求测试台。不创建任何用户可见的信息——服务代理从用户的角度静默地演练某项功能，但使用真实的用户活动触发新功能的模拟活动。

例如，假定我们打算为前面例子中的照片共享网站添加照片编辑功能，可以用如下方法进行摸黑启动：

- 为照片编辑功能创建一个软件切换开关：开或关。（软件切换开关在 2.1.9 节中描述）
- 为照片编辑功能创建一个摸黑启动切换开关：开或关。
- 创建一个样板照片编辑事务，保存到虚拟区域。客户的照片没有被更改，但是在后台编辑照片并将修改后的版本保存到其他地方。
- 修改现有事务，使其在摸黑启动开关“开”时有不同的表现。在这种情况下，照片上传将被修改为在上传发生到 25% 的情况下运行样板照片编辑事务，将上传的照片提供给编辑事务。

25% 的比例只是一个例子——可以采用任何比例。这只代表一个以后可以根据事务数据计算的已知数字。对于真正复杂的服务，从较为保守的数字（如 5%）开始可能是个好主意。使用资源回归分析技术观察新发布功能和哪些类别的资源成本相关联。这将根据实际的生产环境使用情况，帮助你进行第一遍的新功能发布容量规划。相应地调整容量，然后继续摸黑启动，修复找到的所有缺陷并在必要时调整容量。逐步增加摸黑启动事务的采样率，直到达到真实的使用程度。一定要超出某个比例，为自己留有余量。

最终，摸黑启动被关闭，为客户打开新功能。利用对应的切换开关，这可以在不推出新发行版本的情况下完成。在实践中，摸黑启动期间会修复许多缺陷，在此期间几乎等于

升级了好几个新版本。但是,所有这些缺陷应该对客户社区不可见,在完成摸黑启动之后,实际的发布情况将会更好。

案例研究: Facebook Chat 的摸黑启动

“摸黑启动”(Dark launch)一词是2008年Facebook展示用于发布Facebook Chat的技术时提出的。这次发布提出了一个重要的问题:如何在一夜之间将用户量从0增长到7000万而不会造成伸缩性问题。运行中断在这时候将会十分引人注目。在这项功能可见于用户之前很长时间,Facebook页面都被编程为向聊天服务器发起连接,查询状态信息,并在不于页面上绘制任何UI元素的情况下模拟消息发送(Letuchy 2008)。这为Facebook提供了预先找出并解决所有问题的机会。Facebook用户们对Web浏览器发送模拟的聊天消息一无所知,但是他们所提供的测试带来了很大的价值。(11.7节有相关的案例研究)

18.5 缩短配给时间

正如18.1.4节中所讨论的,影响余量需求的因素之一是资源获取和配给的时间。如果可以缩短获取和配给时间,就可以减少余量,相应地减少闲置资本投入数量。此外,获取和配给越快,对需求更改的响应就越快,也是一个重大的竞争优势。

但是,闲置过剩容量也可能是无法快速缩减可用资源的一个结果。放弃增加的容量及其相关成本的能力也是一种竞争优势。有许多方法可以缩减闲置过剩容量,它们往往可以组合以达到更好的效果:

- ❑ **租赁计算机而不是采购。**根据库存的情况可能有较短的获取时间,终止租赁、归还硬件的能力也可能有用。租赁可以用于将容量成本从资本性支出变成运营成本。
- ❑ **使用虚拟资源。**虚拟资源的分配很快,且相关的启动成本很低甚至为零。这些资源也可以快速终止,而账单仅反应实际的资源使用。
- ❑ **改进新硬件资源的订购过程。**预先批准预算决定,建立标准化订单。
- ❑ **缩短安装时间。**配给时间的一部分是从硬件到达货运码头到实际投入使用之间的时间。找出简化机架安装和老化过程的方法。
- ❑ **管理时间。**在新设备到达时第一时间进行安装,不要有闲置的硬件。另外,可以由专人负责安装工作。雇用非系统管理员(“技术员”)进行拆包和系统上架工作。
- ❑ **和供应商合作(供应链管理)。**缩短订购时间。
- ❑ **下多个较小的订单而不是一个巨型订单。**这可以改进系统的并行性。供应商可能会将一个大的订单分解为定期交付并按月记账。从6个月期的大型订单过渡到6个月度,订单和交付在记账、资金成本和人工成本上都可能的好处。
- ❑ **自动化配置。**一旦新硬件上架,很快就可以使用。

18.6 小结

容量规划是确保服务在需要时有足够资源的过程。避免服务因为缺乏容量而中断，同时只增加当时必要的容量，是很有挑战性的。

标准容量规划基于当前使用情况和简单的变化率，假定未来资源需求类似于当前使用量加上两类增长。正常（自然）增长是根据当前趋势预计的。计划增长是营销计划等新活动而预期的增长。附加容量（余量）用于应对短期高峰。根据显示提前期（获得和配置新资源所需的时长）的时间表，可以确定容量计划。通过缩短提前期，容量规划可以变得更加敏捷。

对于小型、成长缓慢和需求简单的网站，标准容量规划就足够了。但是对于大型、快速成长的网站就无能为力了。这些网站需要更高级的技术。

高级容量规划基于核心驱动力、单独资源的容量限制或者精密的数据分析（如相关、回归分析、统计预报模型）。回归分析寻找核心驱动力和资源之间的相关度。预报使用过去的的数据预测未来需求。

对于足够大的网站，容量规划是全职工作，往往由具有技术背景的项目经理完成。有些组织雇佣全职的统计人员建立复杂模型和仪表盘，提供项目经理所需的信息。

好的容量规划模型还可以发现资源需求的意外变化，例如，新软件发行版本意外地需要更多资源。

容量规划是高度数据驱动的，使用过去数据预测未来需求。因此，发布全新服务带来了特殊的挑战。摸黑启动和其他技术使服务在可见于用户之前能够收集准确的数据。

缩短配给新资源的时间可以改进成本效益。配给包括获取、配置和投产新资源。总配给时间被称作提前期。长的提前期会将资金锁定。缩短提前期减少闲置容量，改进财务效率。

容量规划是可靠运营的一个复杂而重要的部分。

练习

1. 描述标准容量规划的工作方式。
2. 描述高级容量规划的工作方式。
3. 对比标准和高级容量规划。何时使用它们最佳？
4. 发布新服务的难点在哪里？
5. 列出你的主要应用程序使用的资源。
6. 使用回归分析确定两个资源之间的相关度。
7. 为服务容量需求创建预报模型。
8. 描述如何为主应用的一个新功能实施摸黑启动。

9. 什么是资源回归，为什么它很重要？

10. 在当前应用发行版本和前一个发行版本之间执行资源回归。讨论你的发现以及对容量规划的意义。

11. 有许多缩短配给时间的途径。哪三种途径对你的环境最有影响？（另外，根据你选择的类别，分类本章列出的方法）

12. 为什么缩短配给时间是很有必要的？

定义良好的 KPI 遵循 SMART 准则：具体（Specific）、可计量（Measurable）、可实现（Achievable）、相关（Relevant）和有期限（Time-phased）。KPI 的定义必须无歧义且不会过于宽泛。可计量意味着可以客观地量化成功或者失败。KPI 可以在合理的情况下实现，与组织（或者项目）的整体成功相关。KPI 是有期限的，意味着规定了相关的时间周期。

目标应该是可计量的，因此人们可以无歧义地确定目标的哪些部分已经实现，或者完全实现。可以计数的事情，如正常运行时间、磁盘空间和本月发布的主要功能数量都是可计量的。可计量目标的例子包括：

□ 为每个用户提供 10T 磁盘空间。

□ 页面加载时间低于 300 毫秒。

□ 本月发布 10 项重大功能。

□ 99.9% 服务器正常运行时间。

一家创业公司制定了一个年度目标：提高客户满意度。这个目标听起来很合理，但很难衡量。客户满意度是一个主观的概念，不同的人可能会有不同的理解。为了将这个目标转化为可计量的 KPI，公司决定采用以下方法：首先，定义客户满意度的衡量标准，例如通过客户调查得分；其次，设定具体的目标值，例如将客户满意度得分提高 10%；最后，规定时间期限，例如在下一财年实现。这样，客户满意度这个目标就变成了一个具体的、可计量的 KPI。

当然，上述目标都是好的。它们都可以转化为可计量目标。但是，如前所述，它衡量的是结果而不是过程。过程变化会影响结果，但过程本身并不直接反映在 KPI 中。KPI 有许多不同的名称。有时候，它们被非正式地称为“关键绩效指标”或“关键成功因素”。在 Google 内部，它们被称为“OKR”（Objectives and Key Results）。

英特尔的 OKR 系统。英特尔使用一个名为 OKR 的系统来管理其目标。这个系统由两个部分组成：目标（Objectives）和关键结果（Key Results）。目标通常是定性的描述，而关键结果则是量化的指标。英特尔的 OKR 系统旨在帮助员工明确目标，并跟踪进度。Google 的 OKR 系统。Google 的 OKR 系统是其著名的目标管理工具。它由拉里·佩奇和杰里·施瓦茨在 2006 年引入。Google 的 OKR 系统旨在帮助员工明确目标，并跟踪进度。Google 的 OKR 系统被认为是目标管理的典范。它的出色解释，可以作为在团队或者企业中采用 Google OKR 系统的教程（Klaauw 2012）。

18.6 小结

容量规划是一个持续的过程，避免服务因为缺乏容量而中断。容量规划是一个持续的过程，避免服务因为缺乏容量而中断。

Chapter 19

第 19 章

建立 KPI

我认识的人中唯一理智的是我的裁缝，他每次见到我的时候都重新量身。其他人都使用旧的尺度，希望我适应他们。

—乔治·萧伯纳

一家创业公司确定，网站的速度对业务的成功很重要。管理层决定，页面加载时间将是决定员工年底加薪幅度的**关键绩效指标**（Key Performance Indicator, KPI）。很快共享机器的服务都得到了专有资源，避免进程间干扰的可能性。只购买最快的机器。许多功能都被推迟了，因为时间都用在代码优化上。到这一年的年底，KPI 指标确认网页具备极快的加载速度。上述的目标达到了，遗憾的是，该公司没有钱加薪，因为它的做法脱离了业务。

计量影响行为。人们在知道自己被计量的时候会改变行为，倾向于找出实现目标的最短路径，这会造成意料之外的副作用。

在本章中，我们谈论的是设定目标和建立 KPI 的明智方法。经理们必须设定目标，推动理想的行为，实现理想的结果，同时最大限度地减少意外的结果。如果采用正确的方法，这会使我们以更高效、更公平的方式管理运营，产生更好的结果。

KPI 的设置可能是经理最重要的工作。人们常常认为，经理有两项职责：设定优先级和提供资源完成优先的工作。设定 KPI 是验证优先级实现的重要手段。

KPI 本身的有效性必须通过在推出前后计量、观察差异加以评估。这可以将管理从松散的猜测变为一组科学方法。我们计量系统的质量、建立或者更改策略，然后再次计量以观察其效果。但是，实际做起来比听起来更困难。

19.1 什么是 KPI

关键绩效指标是用于评估组织或者特定活动成功与否的绩效计量的一种类型。KPI 通常用于激励组织或者团队实现特定目标。

KPI 应该和组织的战略、愿景或者使命直接挂钩。它们通常来自高级管理人员，但是其他层级的管理人员常常也为了自己的目的而创建 KPI，通常作为推进与其相关的 KPI 的一种手段。

定义良好的 KPI 遵循 SMART 准则：具体（Specific）、可计量（Measurable）、可实现（Achievable）、相关（Relevant）和有时限（Time-phrased）。KPI 的定义具体、无歧义且不会过于宽泛。可计量意味着可以客观地量化成功或者失败。KPI 可以在合理的情况下实现，与组织（或者项目）的整体成功相关。KPI 是有时限的，意味着规定了相关的时间周期。

目标应该是可计量的，因此人们可以无歧义地确定目标的哪些部分已经实现，或者完全实现。可以计数的事物，如正常运行时间、磁盘空间和本月发布的主要功能数量都是可计量的。可计量目标的例子包括：

- 为每个用户提供 10T 磁盘空间。
- 页面加载时间低于 300 毫秒。
- 未决的“1 级严重性”缺陷少于 10 个。
- 本月发布 10 项重大功能。
- 99.99% 服务正常运行时间。

不可计量目标无法量化，或者不包含具体的数字化目标。下面是一些例子：

- 更好地掌握 Python 代码的编写。
- 更好地掌握 DR 系统的故障切换。
- 提供更多可用磁盘空间。
- 加快页面加载。

当然，上述目标都是好的。它们都可以转化为可计量目标，但是，如前所述，它们本身不可计量。

KPI 有许多不同的名称。有时候，它们被非正式地称作“指标”，从 KPI 本身就是一种指标的意义上，这是正确的——KPI 是用于驱动组织行为的指标。但是，这就像把橙子称作“水果”，希望人们知道你所指的是特定类型的水果一样。

Intel 的 OKR 系统

Intel 使用一个关联的术语 OKR，是目标与关键结果（objective and key result）的缩写。OKR 往往用于设定个人、团队、部门和公司级别的目标。关键结果常常通过 KPI 计量。缩写词 OKR 因为风险投资人 John Doerr 而流行，他将这个概念带进了 Google。Rick Klau 的视频“[How Google Sets Goals: OKRs](#)”（Google 如何设定目标：OKR）是 OKR 的出色解释，可以作为在团队或者企业中采用 Google OKR 系统的教程（Klau 2012）。

19.2 创建 KPI

创建好的 KPI 需要认真地投入时间和精力。这一过程有许多步骤。首先，我们想象如果目标实现，世界将是什么样子。接下来，确定量化与理想状况之间距离的方法。这可以导出一个或者多个潜在的 KPI。然后，考虑人们与这种激励相匹配的所有可能表现方式。根据上述信息，修订潜在 KPI。重复这些步骤，直到得到最终的 KPI。

在定义正确时，KPI 可以将团队的绩效提高 10-30%。典型工程师的总成本（包括工资、福利和其他费用）每年可能为 20 万美元或者以上。对于有 50 名工程师的组织，这样的改善价值为每年 100 万~300 万美元。大部分面对 300 万美元项目的经理都会花费几天或者几周进行计划，确保项目正确执行。至于投资回报，花费 10 个小时进行这种改进，可以得到 1:1000 或者 1:3000 的回报。谁会拒绝这样的回报？但是 KPI 的创建常常没有经过深思熟虑，意料之外的副作用给上述的好处带来负面的影响。

上面的数字应该成为你发展创建有效 KPI 所需技能的个人动机。更好的 KPI 可能比你所说或所能做的其他事情更为重要。

19.2.1 步骤 1：想象理想状况

暂停一下，想象一下如果目标完美地实现，世界将会怎样。它和今天有什么不同？从最终结果去思考，而不是考虑实现这一结果的过程。训练你的创造力，你将如何描述公司、项目或者服务？资源将如何使用？

可以从中得到启迪的一个地方是在此领域做得很出色的子团队。一般来说，他们没有你所能借鉴的 KPI，但是往往有你希望重现的特质。有时候，这种特质可能是他们的文化，而不是特殊的有形结果。

我们最常看到的错误是，经理们跳过了第一步，直接创建 KPI 方程。走这条捷径是个坏主意。如果不首先决定目标，就不太可能到达那里。

19.2.2 步骤 2：量化与理想的距离

问问自己，我们如何计量现在和理想的距离？我们创建许多可能计量这一距离的候选 KPI。计量指标可能是时间长度、计数和某种事情发生（或者不发生）的次数、一个数量或者任何其他数值。计量应该是真实、可重复的。

例如，如果理想是用户拥有比 PC 原生软件应用更快捷的 Web 体验，计量与理想之间距离的方法就和页面加载时间计量有关。如果理想中的世界没有安全漏洞，计量和理想之间距离的方法就是每个月发现的已知安全漏洞的计数。理想也可能是在特定时限内调查的所有已检出入侵，在这种情况下，我们可以量化从检测到开始调查的时间，也可以量化调查的时长。

19.2.3 步骤 3: 想象行为的变化方式

对于每个潜在的 KPI, 尝试打败它。人们可以满足激励条件的行为方式有哪些? 人们如何最大化自己的个人所得?

将员工“正常表现”的希望放在一边。如果他们遵循你的公式, 就已经在表现了。你不能期望他们理解你的想法和意图。

销售人员就是一个很好的例子。如果他们销售任何产品, 就会立刻分析佣金结构, 找出重点产品, 以最大限度地利用他们的时间。如果以相同的价格、相同的佣金率销售两种产品, 但是其中一种比较容易销售, 另一种产品就不会引起注意。如果销售 10 种产品, 每种的价格、佣金和销售成功率都不同, 他们就会进行数学运算, 找出应该积极销售的产品, 其他产品则被忽略。这不是欺骗行为, 而是遵循提供给他们规则。这就是一些公司将大客户和小客户的销售团队分开的原因。当人们可以和大公司签订价值百万美元的订单时, 谁愿意去理会价值 1000 美元的小客户? 除非大订单的销售周期比小订单长 1000 倍, 否则效率就会更高。没有专门的销售团队, 公司就会错过对小公司的任何销售机会。

类似地, 工程师将会研究所得到的 KPI, 像上面所说的那样遵照执行。同样, 这不是“和体系博弈”, 而是简单地遵循规则。工程师做的就是 KPI 告诉他们要做的事。“和体系博弈”应该是篡改日志或者利用 KPI 为己渔利。不要在人们遵循你所编写规则时与意图相左而感到沮丧或者意外。回到前面的安全性示例, 实现目标的最简单方法就是关掉任何检测安全漏洞的机制。这可能不是预期的反应, 因此, KPI 应该进行修订。

19.2.4 步骤 4: 修订和选择

根据在第 3 步中学习到的知识, 我们可以修订 KPI。可以选择用一个 KPI 代替另一个, 或者回到第 1 步重新开始。

寻求确认和非确认数据。例如, 根据过去或者当前指标计算 KPI。如果这个指标不可用, 用模拟数据测试。

寻求其他人的意见。询问他们在这些 KPI 下将如何表现。你可能因为他们在 KPI 中是否有既得利益而得到不同的答案。如果受访者的工作由 KPI 评判, 根据受访者是偏向于个人所得还是不想得到严苛的评判, 过滤他们的意见。如果受访者从 KPI 目标实现得益, 他们就会有相反的偏向, 但是可能不会与用于颠覆 KPI 的内部过程保持一致。

就尚未公布的 KPI 寻求反馈可能造成关于新 KPI 的流言。正如小孩子的“电话”游戏, 随着消息的传播, 误解会越来越多, 在谣言传播中每个人都会将新 KPI 描述得愈加苛刻。这将伤害士气, 在真正的 KPI 公布时造成令人们困惑的错误信息。最糟糕的情况是 KPI 草案已被抛弃, 但是流言仍在继续。因此, 对潜在 KPI 的讨论应该在理解这些 KPI 可以和谁分享的情况下进行。

有时候, 可以在一个团队或者一个项目上测试 KPI, 然后再将其应用到其他领域。这可以帮助我们得到人们对此有何反应的真实经验。此外, 可以留出一定的试用期, 这样人

们可以预期 KPI 成为政策之前经过一些修订。

在 KPI 成为政策之后，我们就没有太多机会修订了。如果公布 KPI 之后出现了意料之外的负面的副作用，你应该修改 KPI，解决任何缺陷。但是，如果 KPI 不断变化，管理层就会给人留下不知自己在做什么的印象，员工会失去信任。如果规则总是在变化，士气将严重受挫。人们刚刚完成调整自己的行为去适应某个 KPI，它又做出改变，要求反向调整，这使他们觉得“脚下的地毯被抽走了”。

因此，应该花大力气从一开始就保证 KPI 的正确。因为构造良好的 KPI 所带来的好处以及设计不佳的 KPI 可能造成的破坏，这一工作应该优先认真进行。最终的 KPI 不一定是简单的，但是必须容易理解，容易理解的 KPI 就容易遵照执行。

回到我们的安全漏洞示例，最后一个 KPI 草案包含了缺陷：没有同时包含入侵检测系统的覆盖率和检测到入侵数量。因此，我们将其修订为 3 个 KPI：一个反映入侵检测系统在各子网的覆盖率，第二个反映入侵总数，第三个反映调查时长。

19.2.5 步骤 5：部署 KPI

下一步是部署 KPI，或者将其制定为政策。这主要是一个沟通职能。新 KPI 必须传达给负责 KPI 的团队以及关键利益相关方、管理层等。

部署 KPI 意味着使人们了解 KPI，并部署计量 KPI 的机制。KPI 应该在未来可能需要修订的假设下部署，但是这不应该成为创建工作做得不好的借口。

如果可能，用于计算 KPI 的指标应该自动收集。这可以通过监控系统或者从日志中抽取信息来完成。不管用哪一种方法，这一过程不应该要求人为干预。即使你忘记了这些指标，它们也会自动积累。

如果指标不能自动收集，可以采取两种措施。一是找出确保指标人工收集的手段。这可以是自动邮件提醒或者重复的日程表项目。另一种则是开发一种自动化收集机制。没有办法自动收集的指标，只有收集尚未自动化的指标。

一旦收集了数据，就应该创建一个仪表盘。仪表盘是显示 KPI 可视化结果的网页，可能已经关联了指标和有用的信息，有利于深入分析数据。

重要的 KPI 应该有适合于大显示器的第二个仪表盘。在走廊或者其他公共区域安装大型显示器，持续显示这个仪表盘。这种显示应该使用大字体和容易从远处观看的图形，作为 KPI 状态的重要提醒，也可以表现达成 KPI 时的自豪感。

19.3 KPI 示例：机器分配

假定我们开发一个 KPI，评估虚拟机创建过程的质量。这个 KPI 可以应用到公共云服务提供商，作为大型公司内部云服务中 VM 创建团队的指导方针，或者只为了评估团队用于创建自己的 VM 的过程。

19.3.1 第一遍

我们从第一步开始——想象理想状况。在理想世界里，人们将在请求之后立刻得到所需的 VM，没有任何延迟。

在第 2 步（量化与理想的距离）中，我们简单地计量从请求到 VM 创建之间的时长。

在第 3 步（想象行为的变化方式）中，我们对人们与此激励相匹配的所有表现方式进行了头脑风暴。我们预见到许多挑战，挑战之一是人们可能在对“开始时间”的定义上有很大的创造力。这个时间可能是从用户接收到请求的时间，也可能是创建过程实际开始的时间。如果请求是通过服务台请求系统创建单据进行的，处理单据之前的延迟可能非常大。如果请求通过 Web 门户或者 API 进入，它们可能进入队列，按照顺序处理。如果等待时间没有包含在这个指标中，团队的表现可能看起来很好，但是不能真实地反映用户接受的服务。

较为现实的指标来自于从客户角度出发的端到端结果计量。这样做可以启发团队从基于单据的请求系统转移到自助服务门户或者 API。这不仅代替了重新输入单据数据的人工过程，而且确保从一开始就能收集到完成过程所需的信息，因此避免了为收集用户忘记或者不知道需要包含在单据中的信息而进行反复的沟通。

人们还可能在定义请求“完成”时间上表现出创造力。其动机是尽快说明完成时间。VM 的创建时间指的是 VM 管理器分配 RAM、磁盘和其他资源以创建空虚拟机的时间，还是虚拟机加载操作系统的时候？如果另一个团队负责 OS 安装机制，我们可以不管 OS 安装是否成功，在这一工作开始之后就“完成工作”吗？对于用户，安装到一半的失败的 VM 是没有价值的，但是错误的 KPI 可能允许这种情况。

团队可能将这些终点作为合理的结束时间。因此，我们回到第 1 步，使用本章学习到的知识，建立更好的 KPI。

19.3.2 第二遍

我们对最终结果的原始定义非常狭窄，应该将其扩展，将重点放在用户所认为的理想结果上。对于用户来说，最终的结果是 VM 做好准备，可用于用户所预想的目的。

理想状况下，用户总是接收到已经请求的 VM：有合适的名称、规格（RAM、磁盘和 vCPU 的数量）、操作系统等。因此，在我们的理想世界中，从用户那里收集正确的信息，用户接收和请求相符的 VM。

可能有一些障碍会阻挠客户立刻使用机器。这些障碍应该包含在指标中。例如，使用需要 DNS 变化传播的机器、实施访问控制等等。

还要考虑公司资源的使用。用户可以请求无限数量的 VM 吗？谁为他们付账？资源记账往往是引导用户限制资源使用的有效方法。我们将使用这种方法。

因此，我们将理想世界的定义修订为：用户尽可能快地按照请求获得可用的 VM，并确定记账安排。

继续我们的例子，进入第2步，将开始时间定义为从用户那里接收到请求的时候。结束时间为请求者可以使用 VM 的时间。我们可以将“可用”定义为用户能够登录到该机器。这自动包含了 DNS 传播和访问控制，以及我们所不知道的问题。

KPI 草案变成：

创建时间的第 90 个百分位数，创建时间从请求创建开始，到用户可以登录机器为止。

我们使用百分位数代替平均值，因为正如 17.5 节中所讨论的，平均数可能造成误导。一次长时间延误的创建工作可能不公平地毁掉平均数。

理想的世界不会有资源短缺现象。任何时候只要需要新 VM，就有容量可以分配。新容量将及时上线以满足所有请求。我们可以增加一个与容量规划相关的指标，但是将 KPI 保持在高级抽象上更好，而不是从微观上管理服务的具体运行。我们关心的是最终结果。如果容量规划没有做好，由于我们已经构建了 KPI，就必须面对请求被搁置、等待新容量上线的事实。话虽如此，如果容量规划成为问题，我们可以在以后建立专门与容量规划效率关联的 KPI。

在第3步中，我们想象行为的变化方式。如果 KPI 不考虑撤销的请求，我们可以简单地撤销所有时间过长的请求。例如，如果请求通过服务台单据发出，用户没有提供完成任务所需的所有信息，我们可以撤销请求，让用户知道我们在他们创建新单据时需要知道的信息。这是一个糟糕的行为，但是可以提高 KPI。

如果 OS 安装步骤不可靠，需要多次重启，可能会增加总创建时间。为了提高 KPI 数量，运营人员可能简单地撤销失败的工作而不是重新尝试。同样，这有利于 KPI，但是不利于用户。

我们可以通过显著增加 KPI 的复杂度来避免这个漏洞。但是，让人们知道自己处于监控之下，通常是避免不端行为的更简单方法。首先，可以公布客户发起的撤销请求数量。这使管理层能暗中监视欺骗行为。其次，可以私下与团队经理达成一致，不鼓励这种行为。高管人员可以建立高标准环境为此提供帮助，包括只雇佣不容忍此类行为的经理。例如，这些经理会注意到系统日志没有记录是谁撤销请求，并要求改变这种情况，使得 KPI 能够正确计量用户发起的撤销和运营人员发起的撤销。

第4步（修订和选择）得到了如下的 KPI：

创建时间的第 90 个百分位数，从接收到用户请求开始计量，直到用户能够登录到机器。故障之后的人工和自动重试被作为原始请求的一部分计算，而不是计算为单独的请求。完全撤销的请求将被记入日志，如果在一周内超过 5 次由运营人员发起的撤销，或者所有请求中超过 1% 结束于客户发起的撤销，将进行调查。

第5步部署 KPI。KPI 被传达给关键利益相关方。计算 KPI 所需要的计量和计算本身应该自动化并在仪表盘上显示。在我们的例子中，系统可能生成我们需要的指标或者给请求和结束时间加上时间戳，需要某种事后处理关联这两个时间并计算时长。指标收集后应该保存并可用于任何在用数据可视化系统，以便创建图形化仪表盘。

19.3.3 评估 KPI

部署之后几周，应该对初始结果进行审核，找出预期之外的副作用。

在我们的例子中，根据 VM 创建的频率，形成的 KPI 计量指标可能每天、每周或者每月回顾。这些结果可能启发更多指标，以隔离问题领域。VM 创建过程由许多步骤组成。通过在每一步之前计量等待时间以及每步的时长，就可能很容易找到改进的机会。

例如，计量提前期可能揭示单据提交和实际创建开始之间的时间跨度很大，说明自助服务请求系统有很大的好处。提前期可能揭示请求经常因等待新资源安装而被延迟数周，表明容量规划需要改进。收集任务失败和重试次数的有关数据，可能表明 OS 安装过程不稳定，应该加以改进。DNS 传播延迟可能缩短，或者可以用基于映像的安装加快 OS 安装过程。监控网络利用率可能发现超载的链路，如果加以改善，可能加快整体安装速度。通过分析请求类别，可能决定预先创建几个标准规格 VM，在必要时简单地分发。

这些更改都是可行的。通过计量，我们可以预测每一项工作对 KPI 的改进情况。在更改前后收集 KPI 计量，可以计量实际的改进。

19.4 案例研究：错误预算

Google 的工程副总裁 Benjamin Treynor Sloss 透露了一个很成功的 KPI——Google 错误预算。这个 KPI 的目标是在不压制创新的情况下鼓励更高的正常运行时间，在不引起不合理风险的情况下鼓励创新。

19.4.1 相互冲突的目标

开发人员和运营团队之间在历史上存在冲突。开发人员希望启动新功能，运营团队希望稳定。

开发人员的职责就是变化。他们因为新功能（特别是对最终客户有高可见性）而得到报酬。他们更愿意尽快地投产每项功能，以便更快地赢得成就感。管理层最常问他们的问题可能是“何时交付？”。

运营人员则为稳定性负责。他们不希望出现任何故障，不希望因为紧急传呼而度过糟糕的一天，因此他们厌恶风险。如果可能，运营人员会拒绝开发人员投产新发行版本的请求。如果软件不出问题，就不要去修改它。管理层最常向他们提出的问题可能是：“为什么系统下线了？”

系统稳定之后，运营部门宁愿拒绝新软件的发行。但是，从文化上说这是不可接受的。作为替代，为避免问题而建立规则。规则起初很简单：不在周五升级，如果出现问题，我们不应该花费周末进行调试。然后，因为周一人为错误可能增加，所以这一天也被排除。然后，清早和深夜也被排除。在发行之前增加的安全措施越来越多：1% 的测试从可选测试变成必需测试。基本上运营人员从来不直接说“不”，但是堆积了足够的规则，实际上就

是在说“不”。

开发人员不希望代码交付受困，于是开始绕过这些规则。他们用标志翻转隐藏大量未测试代码。在配置文件中编码重要功能，使软件升级不再必要，只需要新的配置。这些变通措施绕过运营的批准，产生更大的风险。

这种情况不是开发人员或者运营团队的错误，而是判定任何运行中断都是坏事的经理们的错误。100%的正常运行时间是对心脏起搏器的要求，而不是对网站的要求。典型用户通过 WiFi 连接到网站，WiFi 本身的可用性就只有 99% 甚至更少。这使追求完美的任何目标一下子就从高处跌落。

19.4.2 统一的目标

对于网站来说，一般来说 4 个 9 (99.99%) 的可用性就足够了。这留出了 0.01% 的停机时间“预算”——每年略小于 1 小时 (52.56 分钟)。这样就创建了 Google 错误预算。这种做法并不寻求完美的正常运行时间，而是为每个季度预算某些不完美的情况。不允许失败，就会压制创新。错误预算鼓励冒险，但是不鼓励粗心大意。

在每个季度的开始，预算被重置为 13 分钟——大约是 90 天的 0.01%。任何不可用的情况将从预算中减去。如果预算没有耗尽，开发人员可以根据自己的意愿发行软件。当预算耗尽，所有发布都被停止。高优先级的安全性修复是个例外。当计数器重置时发行再次开始，再次拥有了 13 分钟的预算。

结果是，运营部门不再处于必须决定是否允许发布的地位。处于那个地位，每次说“不”都会使他们成为“坏人”，开发人员将他们视为“应该打败的敌人”。更重要的是，将运营置于这个地位是不公平的，因为这种关系中天生存在信息不对称。开发人员比运营人员更了解代码，因此更有利于执行测试、判断发行版本的质量。尽管可能不愿意承认，但运营人员没有读心术。

19.4.3 所有人得益

对于开发人员，错误预算提供了他们十分珍视的东西：发行更多软件的机会，激励他们改进可靠性。这种做法鼓励他们更全面地测试发行版本，采用更好的发行方法，投入精力构建改进运营和可靠性的框架。以前这些任务可能被视为对创建新功能的干扰。现在这些任务可以创造推出更多功能的能力。

例如，开发人员可能创建一个框架，使新代码可以更好地测试，或者用更少的精力进行 1% 试验。他们受到鼓励，利用之前没有考虑的现有框架。例如，2.1.3 节中描述的“跛脚鸭”模式的实现可能内建于他们所使用的 Web 框架中，但是他们却没有加以利用。

更重要的是，预算在开发人员团队之间造成同等的压力，要求采用高标准。指定服务的开发通常是许多子团队的成果。每个团队都希望频繁发布产品。但是如果粗心大意，一个团队就可能挥霍掉所有团队的预算。没有人希望成为最后一个采用促进发布成功的技术

或者框架的团队。而且，开发团队之间的信息不对称较少。因此，团队可以为代码评审和其他过程采用高标准。（代码评审在 12.7.6 节中讨论）。

这并不意味着 Google 认为每年停机一个小时没关系。回顾 1.3 节，用户可见服务往往由许多其他服务的输出组成。如果这些服务中有一个不响应，用通用的补白代替遗漏的部分、显示空白或者使用 2.1.10 节中描述的其他优雅降级技术，这一组合仍然可以继续工作。

这个 KPI 已经成功地改进了 Google 的可用性，与此同时保持了开发人员和运营人员优先事项的统一，帮助他们一起工作。它使运营部门摆脱了拒绝发行的“坏家伙”角色，激励开发人员平衡新功能添加和运营过程改进的时间。这个 KPI 很容易解释，因为可用性已经得到紧密的监控，也很容易实现。结果是，所有 Google 服务都从中获益。

19.5 小结

使用 KPI 管理是彻底抛弃传统 IT 管理的一个举措。这个方法很有效，因为 KPI 设定目标，允许你所雇佣的能干的员工确定实现目标的途径。这些人距离任务更近，对技术细节更为了解，因此更适合于发明实现目标的方法。

建立有效的 KPI 很困难——也应该困难。在措施得当时，KPI 可以获得巨大的投资回报。一本万利的事没有容易做的。KPI 应该具体、可计量、可实现、相关和有期限。当人们按照字面遵循规则而不按照你的意图行事时，编写质量不佳的 KPI 会造成意外的结果。彻底想清 KPI 所有的诠释方法，以及人们为了改进自己在这—指标上的表现所采取的措施，是很重要的。

预先花费时间解释和修改 KPI，确保 KPI 最有可能给出你所需要的结果。不要使 KPI 过于复杂，如果怀疑 KPI 可能触发不利的行为，就要计量和发布这些行为，确保经理们知道如何监视和阻止这些行为。

记住，KPI 应该是可实现的。100% 的可用性是无法实现的，除非“可用性”的定义做了调整，覆盖计划之外的范围。但是，4 个 9（99.99%）是可以实现的。

Google 的错误预算 KPI 成功地使用该目标实现了理想的结果：稳定的服务，创新的新功能频繁部署。这是好的 KPI 惠及所有人的一个出色范例。

练习

1. 什么是 KPI？
2. 什么是 SMART 准则？简单描述每一条。
3. 举出 KPI 意外副作用的例子。
4. 创建 KPI 有哪些步骤？评估哪些步骤最难，并证明它们很难的原因。
5. 你在环境中跟踪哪些 KPI，为什么？

6. 创建一个有效的 KPI 以评估环境中的一个服务。创建之后，让另外 3 个人告诉你如何最大化他们的个人所得。修订 KPI，报告理想状态（第 1 步）、初始 KPI、人们的反应和最终 KPI。
7. 你所创建的 KPI 如何满足 SMART 的各条准则？
8. 通过 KPI 管理时，为什么人们会遵循书面的意义而不遵循其实际意图？这是好还是坏？
9. 19.3 节中的例子总是讨论基于客户角度的端到端时间计量。计量从 VM 创建开始到 VM 可用之间的时间有没有价值？
10. 如果发出请求之后，请求者的经理必须批准请求，你将如何修改 19.3 节中创建的 KPI？

19.4.2 统一的目标

对于网站来说，一般来说 4 个 9（99.99%）的可用性就足够了。这留出了 5.26 分钟的时间“预算”——每星期小于 1 小时（52.56 分钟）。这样就创建了 Google 错误预算。这种预算并不完美，因为预算这个概念本身就是一个近似值，而且它假设网站的可用性是恒定的。然而，在现实世界中，网站的可用性是波动的。例如，在节假日期间，网站的流量可能会增加，这可能会导致网站的可用性下降。因此，Google 错误预算只是一个粗略的估计，而不是一个精确的指标。

在每个季度的开始，预算被重置为 13 分钟。这个预算是基于历史数据计算出来的。如果网站的可用性超过了预算，那么网站的经理就会收到警告。如果网站的可用性低于预算，那么网站的经理就会收到警告。这个预算的目的是为了确保网站的可用性始终处于一个合理的水平。如果网站的可用性超过了预算，那么网站的经理就可以放心了。如果网站的可用性低于预算，那么网站的经理就需要采取行动了。这个预算的目的是为了确保网站的可用性始终处于一个合理的水平。

19.4.3 所有人得益

Google 错误预算的目的是为了确保网站的可用性始终处于一个合理的水平。如果网站的可用性超过了预算，那么网站的经理就可以放心了。如果网站的可用性低于预算，那么网站的经理就需要采取行动了。这个预算的目的是为了确保网站的可用性始终处于一个合理的水平。

例如，开发人员可能创建一个框架，使新代码可以更好地进行测试，或者用更少的精力进行 1% 测试。他们受到鼓励，利用之前没有考虑的现有框架。例如，在 19.3 节中，我们看到了“脚鸭”模式的实现可能内建于他们所使用的 Web 框架中。这个模式的实现可能内建于他们所使用的 Web 框架中。

更重要的是，预算在开发人员团队之间造成同等的压力。在 Google，每个团队都有自己的预算。如果团队的预算被耗尽，那么团队的经理就会收到警告。如果团队的预算没有被耗尽，那么团队的经理就可以放心了。这个预算的目的是为了确保每个团队的可用性始终处于一个合理的水平。

例如，开发人员可能创建一个框架，使新代码可以更好地进行测试，或者用更少的精力进行 1% 测试。他们受到鼓励，利用之前没有考虑的现有框架。例如，在 19.3 节中，我们看到了“脚鸭”模式的实现可能内建于他们所使用的 Web 框架中。这个模式的实现可能内建于他们所使用的 Web 框架中。



第20章 Chapter 20

卓越运营

只有擅长于日常运营的公司才能抓住特别的机遇。

——马塞尔·特列斯

本章是关于服务运营质量的计量或者评估的，提出了评估工具并给出如何用这些工具评估单独服务、某个团队或者多个团队的示例。

在第19章中，我们讨论了如何创建KPI，推动预期的行为，实现特定目标。本章描述一个评估过程正式程度和优化程度的评价系统——过程是临时、正式还是主动优化的。这种评估不同于KPI，在更普遍的水平上计量团队，更注重团队之间或者团队内各个服务的可比性。

这种评估有助于确定改进的领域。我们可以进行更改、重新评估并计量改进。如果定期进行，就可以创造一个持续改进的环境。

20.1 卓越运营是什么样子的

出色的系统管理员是什么样子的？正如艺术和文学，当你见到的时候知道它是什么样子，但是却难以定义。这种模糊性使得系统管理团队的表现优劣难以量化。

表现出色的组织有顺畅的运营、精心设计的策略和方法，以及行为准则。它们达到或者超出客户的需求，并用满足未来需求的创新取悦客户，这些创新往往先于需求出现。这种组织的计划、运营、服务提供和成本处理或分摊对客户都是透明的。大部分客户都感到很快乐。即使不满意的客户也觉得他们的声音会被听到，有升级问题的渠道。每个人都觉得运营组织在推动公司向前发展。运营组织的资金筹措反映了组织对工作的合理预算。组

织展现其成功，更重要的是，在讨论自己的错误时坦诚而直率。这种组织不断改进，运行中断和升级的问题促成了减少问题未来发生次数的行动计划。世界不断改变，组织引入新技术改进内部工作，以及他们所提供的服务。

运营组织似乎可以归入 3 个宽泛的类别或者阶层：出色、希望出色和不知道出色为何物的。

我们估计有 5%~10% 的运营团队可以归入第一类。它们知道行业的最佳实践并加以应用，有些组织甚至发明新的方法。接下来的 25%~30% 知道最佳实践的存在，但是仍在寻求采用这些方法的路上。剩下的大部分组织甚至不知道这些最佳实践的存在。

科幻小说家 William Gibson 曾经说过，“未来就在眼前——只是分布得不太均匀。”同样，如何成为出色系统管理团队的知识也已经摆在眼前——只是分布得不太均匀。

20.2 如何计量卓越的程度

运营团队质量的计量极其困难。运营的其他方面容易计量，例如，团队的大小可以通过计算成员人数、提供的服务数量和每年的支出来计量。规模可以通过计算机器数量、存储总量和总使用带宽等计量。我们可以用成本比率计量效率。

可惜，质量没那么容易计量。

想象可能需要计量质量的一个时刻。想象我们有一种标准方法，以简单的值（0~1000）来为运营团队的质量打分。再想象一下，我们可以通过某种魔法为世界上的所有运营组织排名。

如果我们可以做到，将这些组织从“最佳”到“最差”排序，那么学习的潜力是惊人的。我们可以观察前 50% 的组织 and 后 50% 的组织之间的差别。另外，某个组织可以研究更高排名的组织，获得改进工作的灵感。

遗憾的是，没有这样的单一计量指标。运营太复杂了，因此，计量（或者评估）必须反映这种复杂性。评估听起来更像我们在学校里得到的等级，但是概念上差异很大。学生评估考察的是单个学生的学习和表现。等级评估学习，还要加入出勤、参与和成就。评估的重点更突出。

对服务的评估基于与过程成熟度相关的特定条件，而不是对服务是否流行、是否具备高可用性或者是否快速评估。并不是所有服务都必须是流行、高可用或者快速的。相反，所有服务都需要好的过程，以实现它们的任何目标。因此，我们评估过程是因为好的过程是通往成功的路线图。

20.3 评估方法论

这种评估方法是自底向上的评估。服务按照 8 个属性评估，这些属性称作运营职责

(Operational Responsibility, OR)。每个 OR 按照 1~5 级评估, 5 为最佳。如果评估定期进行, 人们可以随时看到进展。可以使用加权平均数, 累计 8 项单独评估得出代表服务水平的单一数字。

团队对每个服务进行这种评估。用所提供服务的加权平均成绩可以评估一个团队。然后, 用堆栈排名比较各个团队。团队可以找出问题领域, 寻求提高排名。可以确定高排名团队的最佳实践并在各个团队中分享。8 个核心 OR 面向服务管理, 不适合于事务性的 IT 服务, 如服务台或者其他前台、1 层或其他面向客户的服务中心。

20.3.1 运营职责

我们已经确定了运营职责的 8 个宽泛类别, 大部分服务都具备这些运营职责。对于特定的服务, 有些运营职责可能较为重要, 有些则较不重要。你的团队在累计时可以使用加权平均数, 强调或者不强调某些 OR。团队也可以在必要时选择增加更多 OR。

常见的 8 种运营职责如下:

- ❑ **常规任务 (RT)**: 常规、非紧急运营任务的处理方式。也就是, 如何接收、排队、分布、处理和验证工作, 以及如何安排和执行定期任务。
- ❑ **紧急响应 (ER)**: 运行中断和灾难如何处理。这包括运行中断期间和事后执行的技术和非技术过程 (响应和补救)。这是第 6、14、15 章的内容。
- ❑ **监控和指标 (MM)**: 收集和使用数据做出决策。监控收集关于系统的数据。指标使用数据计量性能的可量化部分。这是第 16、17 和 19 章的内容。
- ❑ **容量规划 (CP)**: 确定未来资源需求。容量规划涉及理解每个增长单位所需资源数量的技术工作, 以及预算、预测和供应链管理等非技术方面。这是第 18 章的内容。
- ❑ **更改管理 (CM)**: 管理服务在一段时间内的有意更改方式。这包括服务交付平台和使用它创建、交付和投产新应用及基础架构软件的方式。更改管理包括固件升级、网络更改和 OS 配置管理。
- ❑ **新产品推出与删除 (NPI/NPR)**: 决定如何在环境中推出新产品和服务, 以及如何废弃及删除这些产品及服务。这是一个协调职能。在环境中推出和删除服务或产品涉及多个团队。删除服务涉及当前用户的跟踪, 以及从将要被淘汰的服务中迁移的管理。例如, 这可能包括协调因为生态系统中引入新硬件品牌而受到影响的所有团队、发布全新服务或者退役旧硬件平台的所有实例。
- ❑ **服务部署和退役 (SDD)**: 确定现有服务的实例如何创建以及如何关闭 (退役)。在服务于某个环境中推出时, 会被部署许多次。在为其目的服务之后, 这些部署将退役。例子包括启动一个新数据中心、添加服务副本或者添加数据库副本。
- ❑ **性能与效率 (PE)**: 计量服务的表现以及资源利用的成本效益。运行中的服务必须有良好的性能且不浪费资源。这方面的例子包括管理利用率或者电源效率以及相关的成本。

NPI、SDD 和 CM 之间的差别很细微。NPI 是第一次发布的方式，这是所有相关过程的非技术协调职能，而这些过程中许多都是技术性的。SDD 是部署现有项目新实例的技术性过程，这些项目可能是机器、服务器或者服务。SDD 可能包含预算批准之类的非技术性过程，但是这些过程都是用于支持技术目标的。CM 是升级和更改的管理方式，可能使用软件部署平台或者企业风格的更改管理审核委员会。

上述术语的更详细描述可以参见附录 A，附录 A 中还包含了本书某些章节中特有的其他 OR。

20.3.2 评估级别

评级（评估）根据能力成熟度模型（Capability Maturity Model, CMM）给予 1~5 的评分。CMM 是设计用于计量能力或者职责成熟度的工具。

“成熟度”一词表示运营中某个方面实践的正式或者非正式程度。正式程度的范围从临时过程，到有文档记录的过程，到计量过程结果，到根据计量主动改进系统。

CMM 定义 5 个级别：

- ❑ 级别 1，初始：有时候称作“混沌”（Chaotic）。这是新过程或者无文档过程的起点。过程是临时的，依赖个人英雄主义。
- ❑ 级别 2，可重复：过程至少有充足的文档记录，可以重复得到相同的结果。
- ❑ 级别 3，已定义：过程的角色和职责已经定义并确认。
- ❑ 级别 4，已管理：过程根据达成共识的指标量化管理。
- ❑ 级别 5，优化：过程管理包含了有为之的过程优化 / 改进。

级别 1：初始

在这个级别上，过程是临时的。结果不一致，不同的人以不同方式完成任务，结果通常略有不同。过程没有记录文档。工作没有进行跟踪，请求常常丢失。团队无法精确估计任务所需的时长。客户和合作伙伴可能对接受的服务感到满意，但是此时没有基于证据的确认。

级别 2：可重复

在这个级别上，过程从临时变为可重复。步骤经过某种方式的定义，不同人可以遵循这些步骤获得相同的结果。过程文档没有遗漏的步骤，最终结果相对一致。这不是说不会发生错误，毕竟没有任何东西是完美的。因为这一级别的计量很少，我们可能不知道工作出现缺陷的频率。

级别 3：已定义

在这个级别上，过程的角色和职责已经定义和确认。在前一级别，我们知道需要完成的工作。在这一级别，我们知道谁负责执行任务，并且定义了正确性的计量方法。正确性计量可能没有收集，但是至少所涉人员知道这些计量的定义。每个步骤有一系列检查，在

错误发生时发现它们，而不是等到过程结束才发现错误。重复劳动被最大限度地减少。如果我们需提高生产率，可以复制这一过程或者增加更多容量。

级别4：已管理

在这个级别上，过程的各个方面都受到计量。每一步所花的时间得到计量，包括等待过程开始所涉及的那部分时间。计量指标包括每个月执行过程的频率，无出错执行过程的频率，以及需要特殊处理和临时过程的异常情况的出现次数。变化也得到计量。这些计量自动收集，有一个仪表盘显示所有计量指标。瓶颈很容易发现，事后剖析在异常发生之后的一段时间内发布。异常情况进行分类并定期审核。过程更改请求用计量数据说明问题，加以证明。容量需求提前做出预测。

级别5：优化

在这个级别，计量和指标用于优化整个系统。改进在前面几个级别上都已经进行，但是这里所做的改进是由计量提示的，在更改之后，根据新的计量手段评估成功。对每一步时长的分析将暴露瓶颈和延迟。计量表明逐月的改进。我们可以对系统进行压力测试以发现有问题的地方，加以修复，然后在新的水平上运行系统。

已定义和已管理的对比

人们往往很难概念化第3级和第4级之间的微妙差别。考虑评估负载平衡器服务特性的例子。负载平衡器可用于增大容量或者弹性（或者两者兼有）。弹性要求系统运行时具备足够的备用容量，以承受后端的故障。（参见6.3节）

如果有一个将负载平衡器用于弹性的书面策略，这是第3级的行为。如果存在用于确定当前冗余级别（ $N+0$ ， $N+1$ 等）的监控手段，这是第4级的行为。在这种情况下，第4级显然更难以实现，因为需要付出很大努力才能够确定后端的最大容量，这是知道可用备用容量所必需的。

令人惊讶的是，我们发现许多情况下对负载平衡器的使用方式没有清晰的理解，或者团队成员之间对负载平衡器的角色是增加容量还是增强弹性没有达成共识。这就是第1级的表现。

20.3.3 评估问题和匹配属性

要评估运营职责，就要描述该领域的当前实践。根据这一描述，评估描述当前实践的级别。

为了帮助实施这一过程，我们已经开发了一个标准问题集，帮助你形成描述。

我们还为每个级别开发了一组**匹配属性**（Look-for）。匹配属性是常见于特定级别服务或者组织的行为、指标或者结果。换言之，就是“这个级别是什么样子的”。如果你阅读一个描述，它“听起来像在谈论我工作的环境”，这就很好地说明你的组织在该服务上处于这

个级别。

例如，在常规任务中，第 1 级没有常见运营任务的剧本，也没有这些任务的定义列表。在第 3 级，这些任务已经定义并记入文档。

匹配属性不是检查列表。人们不一定要展现该级别中需要评估的每个匹配属性。有些匹配属性仅适用于某些情况或者服务。匹配属性就是简单的信号和指标，而不是遵循的步骤或者寻求的成果。

附录 A 列出了每个级别运营职责、问题和匹配属性。花点时间看看并研究其中的内容。我们等着你。

20.4 服务评估

为了将评估过程付诸行动，工作人员必须定期在每个重要服务或者他们提供的相关服务组上执行评估。评估用于暴露需要改进的领域。团队就解决这些问题的方法展开头脑风暴，选择一定数量的项目以解决最高优先级的问题。这些项目将成为下一季度的任务。

然后，团队重复过程，评估服务。评估给新的项目带来启发。执行这些项目，再评估服务。然后再次开始上述过程。这样，团队可以从一个用于确定项目的结构中获益。他们总是知道应该做的事情。

20.4.1 确定评估的内容

首先，找出组织提供的重要服务。基于 Web 的服务可以找出服务的重要组件（软件组件提供的重大功能组）和基础设施服务（如网络、电源、散热和互联网访问）。企业 IT 组织可以找出为公司提供的重要应用程序（例如电子邮件、文件存储、集中化计算场、桌面 / 便携机群管理）以及 DNS、DHCP、ActiveDirectory/LDAP、NTP 等基础设施服务。较小的网站可以将服务组合起来（DNS、DHCP 和 ActiveDirectory/LDAP 可以组合为“名称服务”），较大的网站可将每个单独组件视为单独的服务。选用其中一种方法，构造重要服务列表。

对每个服务，评估服务的 8 个运营职责。附录 A 中的每个小节列出有助于这种评估的问题。这些问题是通用的，应该适用于大部分服务。你可能希望增加适合于你的组织或者特定服务的其他问题。重要的是对每次评估使用相同的问题，这样数字才有可比性。制作一个参考文档，列出使用的问题。

20.4.2 评估每个服务

在每次评估期间，记录评估分数（1~5）并记下评估的依据。通常，这些记录以问题答案的方式存在。

图 20.1 展示了一个用于跟踪服务评估的电子表格示例。第一列列出 8 个运营职责。其他列各代表一个评估期间。为每个服务使用不同的子表。使用电子表格的“插入注释”功

能记录证明评估值的依据。

按照对服务的重要性排序运营职责是个好主意。例如，不太频繁增长的服务在容量规划中所受的关注较少，所以这个职责可以在最后列出。用红、橙、黄、绿、蓝色的方块分别代表1~5。这给出了可视化表示，并创建了一个“热图”，用颜色的变化显示进展。

对于服务负责人，这种工具是帮助评估服务和跟踪进展的好手段。

服务A	12/2014	1/2015	2/2015	3/2015	4/2015
常规响应	1	2	2	2	3
紧急响应	3	3	4	4	4
监控和指标	1	1	2	2	2
容量规划	1	1	1	1	2
生命周期管理	2	2	2	3	2
新服务推出与删除	1	2	2	3	3
服务部署和退役	2	2	3	3	4
资源效率	1	1	1	2	2

图 20.1 服务评估

20.4.3 比较不同服务的结果

图 20.2 展示了一个用于累计评估值，比较不同服务的电子表格示例。服务应该按照重要性顺序列出。每个数字代表服务的 8 个评估值。这些数字可以加权平均或者简单地算术取“模”（最常见的数字）。不管你使用哪一种方法，都要保持一致。

Team 1	2014Q1	2015Q1	2015Q2	2015Q3	2015Q4
服务A	1	2	2	2	1
服务B	3	4	4	4	5
服务C	1	2	2	3	3

图 20.2 得出对团队的评估

两个电子表格都应该由团队集体填写。经理的任务是保证每个人在精确性上保持高的标准。经理不应该自己进行评估，提供给团队意料之外的结果。自我评估有固有的激励因素，由经理评级在最好的情况下也会带来消极影响。

20.4.4 根据结果采取行动

现在，我们可以确定哪些服务最需要改进，并识别每个服务中的问题领域。

确定哪些项目将修复评估得分很低且改进有很大影响的领域。也就是说，将焦点放在最重要服务上最需要改进的地方。

对大部分服务来说，第3级往往足够了。除非服务直接产生收入或者很高的要求，否则通常应该在大部分职责上实现“稳定的第3级”，然后再扩大项目范围，实现第4级或者第5级。

在下一季度进行所选择项目的工作。在下一季度的开始，重复评估，更新电子表格，并重复该过程。

20.4.5 评估和项目计划的频率

评估的频率和项目计划周期不一定要相同。为每月的评估和每季度的项目周期做计划。月度评估是跟踪进度的较好频率。如果在该时间段内发生的可计量进展很少，更频繁的评估会浪费时间。频率较低的评估难以快速发现新的问题。

项目计划周期可以定为一个季度，这是为了与绩效评定周期保持同步，也可能因为3个月是完成项目、展示成果的合理时间。如果项目选择的频率太低，可能减慢更改的节奏。如果项目周期过于频繁，可能不利于有意义的大项目。

评估之间的时隙越长，带来的负担越大。如果周期是一个月，一两个小时的会议通常就能完成评估。如果周期是一年，评估就变成一个“大买卖”，可能需要在一周内停下所有工作，准备长达一整天的评估会议，分析团队的每个方面。这种年度评估令人害怕、毫无益处且令人厌烦，常常变成官僚主义、浪费时间的会议。这是小批次更好的另一个例证。

20.5 组织评估

由于高管或者经理负责许多服务，这种评估系统是指导团队方向和跟踪进展的好方法。这些评估还能够向团队清晰地传达预期。

负责服务的人应该作为一个小组进行评估。当人们被允许进行自己的评估时，会激励他们承担相关的改进项目。你将惊讶地看到，人们在看到低的评估值时，常常会急于抓住机会，启动项目解决相关的问题，甚至抢在你提出建议之前。

作为经理，你的任务是在团队完成评估时在精确性和一致性上保持高标准。所有服务和所有团队使用的标题应该一致。如果使用附录A中的评估问题和匹配规则，可以对其进

行修订,更好地适应你的组织。

较大型的组织有许多运营团队。对所有团队使用这种评估工具能够发现需要资源和注意的问题的领域,跟踪进展并推进健康的竞争。负责多个团队的高级管理人员应该一起工作,确保标题保持一致。

图 20.3 展示了可用于累计许多可比团队评估值的电子表格示例。注意,团队 1、4 和 5 随着时间的推移一致地改进,团队 4 的进展已经变慢。团队 3 经历了糟糕的一年,刚刚开始取得进展。这个团队可能需要额外注意,确保进展的持续。团队 2 的进展不稳定,在第 3 级和第 4 级之间徘徊,呈缓慢的上升趋势。

	2014Q1	2015Q1	2015Q2	2015Q3	2015Q4
团队1	1	2	2	3	3
团队2	3	4	3	4	5
团队3	1	1	1	1	2
团队4	1	1	2	2	2
团队5	2	3	3	3	4

图 20.3 用累计值比较各个团队

绩效优良的员工将把这种对比看作自豪感的来源或者对更出色工作的启发。得到高评估值的团队应该受到激励,分享他们的知识和最佳实践。

20.6 提高级别

每个 CMM 级别以下一个级别作为基础。第 2 级包含第 3 级的种子,第 3 级包含第 4 级的种子。因此,组织按照顺序通过这些级别,不能忽略任何一个级别。

在所有服务的所有运营职责上都达到第 5 级并不重要。对于用得很少、优先级很低的服务,实现第 5 级是浪费资源。实际上,从专业的角度看,在有更重要的服务需要改进的时候,将必要的资源花在实现低优先级服务第 5 级评估上,是一种草率的做法。

第 3 级评估对于大部分服务已经足够好了。高于这个级别的评估应该留给高优先级服务,如产生收入的服务或者有特别要求的服务。

在设置组织目标时,重点是解决问题而不是实现特定的评估级别。也就是说,不要设定目标,为提高评估值而改进运营职责,这是本末倒置的做法。相反,应该找出问题,设计出解决方案,通过评估值是否提高计量成功或者失败。两者之间的差别很细微,但是

很重要。

设定提高评估值的目标会推动错误的行为。它鼓励人们创建定点解决方案，这种方案不能解决更大的问题，或者只是为了获得更好的评估值而解决容易解决的非重点问题。这就像根据学生得到的分数支付教师的工资，这样的体系将会使考试越来越容易，所有学生都得到 A+。

因此，设定将所有服务的评估值都升级到特定级别的目标同样是错误。工程师的时间是稀缺资源。推动每个组织职责获得高评估值会造成这样的情况：为了提高平均数，这些稀缺资源被花费在低优先级的服务上。这种追求造成官僚主义的镣铐，最终会扼杀组织。绝不应该这么做，这是你希望竞争对手掉入的陷阱。

这一点怎么强调都不为过。如果管理层设置了在所有服务的所有组织职责上达到某个评估级别的企业、部门或者组织目标，你应该让他们看看本书的这一小节，告诉他们这种计划是取祸之道。

同样，绝不应该根据一项评估的结果决定升级和奖金。这会阻止人们加入需要最大帮助的项目。相反，应该鼓励最好的人加入最能发挥自身水平的项目。提供完成出色工作的机会，最能够激发工程师的积极性。对评估值提高的最佳奖赏不是金钱，而是在最感兴趣的项目上工作的机会，或者通过与其他团队分享知识而获得声誉。我们也不建议根据评估级别的提高而进行奖励，因为在面对困境时保持特定级别往往是一项重大的成就。

20.7 开始着手

向团队推介这个评估系统可能是个挑战。大部分团队不习惯于在这种数据驱动评估环境中工作。应该将其作为自改进工具推出和使用——这是帮助团队追求最佳状态的手段。

首先，团队应该列举所提供的主要服务。例如，负责大网站的团队可能确定每个 Web 资产是一个服务、每个内部 API 是一个服务、用于提供服务的公共平台是一个服务。可能存在多个平台，在这种情况下每个平台都被视为一个服务。最后，基础设施本身常常也被作为一个服务评估。

评估应该根据定期、可重复的日程表进行。每个月的第一个周一是常见的选择。团队召开会议进行每个服务的自我评估。管理层的角色是维持高标准，确保各个服务和团队的一致性。管理层可以设定某些 OR 评估的全局标准。例如，可能有一个企业更改管理策略，与这个策略的依从性评估应该对于所有服务、所有团队都是相同的。

应该使用 8 个核心 OR 评估所有服务。附录 A 包含了这 8 个运营职责的细节，以及评估期间帮助团队理解 OR 的所问的问题。这些问题之后是描述各种级别典型行为的匹配属性。匹配属性是服务运营于特定级别的指标。它们不是所要模仿行为的检查列表。不要试图实现每个匹配属性。它们是指标，而非需求或者检查列表。不是每个匹配属性都适合于所有服务。

除了8个核心OR外,附录A列出了特定于指定服务的其他OR。它们是可选的,作为服务特定OR的例子,组织可以从中选择,添加到8个核心OR中。组织还可以选择为自己的特殊需求发明OR。

随着月度评估的进展,一段时间的变化应该很明显。评估的结果帮助团队确定项目优先级。经过一段时间,本章描述的累计分数可以用于对比服务或者团队。

20.8 小结

本章中,我们讨论如何评估运营质量,以及如何使用该评估结果推动改进。

系统管理质量的计量很复杂。因此,对于每个服务我们评估8种不同的特质,称作运营职责:常规任务(正常的非紧急任务如何进行)、紧急响应(运行中断和其他紧急情况如何处理)、监控与指标(用于决策的数据的收集)、容量规划(确定未来资源需求)、更改管理(从诞生到结束,服务如何进行有目的的更改)、新服务推出与删除(新产品、硬件或者服务如何引入环境,如何删除)、服务部署与退役(现有服务的实例如何创建和退役)以及性能与效率(资源使用的成本效益)。

每个运营职责按照5个级别进行评估,反映了软件工程中使用的的能力成熟度模型(CMM)级别。CMM是评估过程的一组成熟度级别:初始(临时)、可重复(文档化、自动化)、已定义(角色和职责达成一致)、已管理(决策由数据驱动)和优化(进行改进并计量结果)。CMM的前三级最为重要,其他级别往往只对高价值服务有必要。

服务的8个职责单独评估。这些评估值累计起来成为服务的单一评估值。团队通常负责许多服务,单独服务的评估值累计起来评估团队,由于每个服务的重要性不同,通常采用加权平均数。这些评估值可以用于为团队排名。

通过定期评估,可以跟踪服务、团队或者组织级别的进展。应该鼓励高排名的团队分享最佳实践,供其他团队采用。

评估的目标是改进并通过观察评估值的变化计量改进的效果,而不应该是提高评估值或者在一组服务中获得平均的评估值。

利用评估推动IT中的决策使我们更接近系统管理的科学管理系统,并使我们摆脱“第六感”和直觉。CMM前三个级别的重要性在于帮助我们摆脱临时过程和个人英雄主义,创建更高效、更高质量的可重复过程。

练习

1. 评估运营部门工作质量或者比较不同运营团队为什么很困难?
2. 描述能力成熟度模型(CMM)的5个级别。
3. 按照重要性(从最重要到最不重要)排列8个“运营职责”,说明排名的依据。

4. 评估在何种情况下会变得过分官僚，如何才能避免？
5. 为什么建议在累计服务评估值时采用算术模数，而在累计团队评估值时使用加权平均数？
6. 累计评估值可以采用哪些其他方式？讨论其优缺点。
7. 选择你负责的一个服务。根据 CMM 级别对每个运营职责进行评估。
8. 对你不负责、但是熟悉其的 IT 人员的服务进行评估。与这些 IT 人员面谈，根据 CMM 级别为每个运营职责进行评估。
9. 比较练习 7 和 8 中的经验。
10. 8 个运营职责可能不适合于所有公司的所有服务。对于你所参与的服务，打算进行哪些修改和增减？说出你的依据。
11. 评估周期过于频繁或者不够频繁都有缺点。每周一次和每年一次都有哪些优点和不足？
12. 本章建议不要设定实现特定 CMM 级别的目标。联系个人经验找出其中的原因（IT 内部或者外部）。

20.7 开始着手

要评估运营职责，首先要明确运营职责的定义。运营职责是指那些与 IT 服务交付相关的活动。运营职责的定义应该基于服务交付的实际情况，而不是基于理论上的完美。运营职责的定义应该包括以下几个方面：

- 1. 运营职责的定义应该基于服务交付的实际情况，而不是基于理论上的完美。
- 2. 运营职责的定义应该包括以下几个方面：

运营职责的定义应该基于服务交付的实际情况，而不是基于理论上的完美。运营职责的定义应该包括以下几个方面：

应该使用 8 个核心 OR 评估所有服务。附录 A 包含了这 8 个运营职责的细节，以及评估期间帮助团队理解 OR 的所问的问题。这些问题之后是描述各种级别典型行为的匹配属性。匹配属性是服务运营于特定 CMM 级别时应该表现出的行为。匹配属性是服务运营于特定 CMM 级别时应该表现出的行为。匹配属性是服务运营于特定 CMM 级别时应该表现出的行为。

其他运营职责

重 次	运营职责	重 次	运营职责
服务交付：构建阶段	服务交付：部署阶段	减少“苦活”	灾难准备

第三部分 Part 3

附 录

评估问题样板

□ 常见的定期运营任务有哪些？

□ 常见紧急任务有哪些脚本？

□ 常规请求的 SLA 是什么？

□ 如何识别新脚本的需求？谁编写新脚本？谁编辑现有脚本？

□ 如何接收和跟踪用户请求？

□ 常见用户请求有没有脚本？

□ 用户请求不包含在脚本的情况发生的频率如何？

□ 用户如何要求我们进行支持？（在线和物理位置）

□ 用户如何知道怎样要求我们的支持？

■ 附录 A 评估

■ 附录 B 分布式计算和云的起源及未来

■ 附录 C 伸缩性术语和概念

■ 附录 D 模板和示例

■ 附录 E 推荐读物

.....

第 1 级：初始

□ 没有脚本，或者脚本已经过时

控制与效率 (PE)

次 页

表附录 A 评估

4. 评估在何种情况下会有得分分布不均，如何才能避免？

5. 为什么建议采用算术均值，而在累计团队评估值时使用加权平均数？

6. 累计评估值可以采用其他方式？讨论其优缺点。

7. 选择你负责的运营任务，根据 CMM 级别对每个运营职责进行评估。与这些熟悉的 IT 人员的服务进行评估。与这些人一起讨论他们的评估结果。

Appendix A

附录 A

评估

本附录包含各种运营职责（Operational Responsibility, OR）的评估问题和匹配属性。第 20 章是本附录的操作指南。入门建议参见 20.7 节。

评估级别

第 1 级：初始 / 混沌	临时，依赖个人英雄主义
第 2 级：可重复	可重复结果
第 3 级：已定义	职责已定义 / 确认
第 4 级：已管理	量化管理指标
第 5 级：优化	有意识的优化 / 改进

核心运营职责

页次	运营职责	所在章
330	常规任务 (RT)	12、14
332	紧急响应 (ER)	6、14、15
335	监控与指标 (MM)	16、17、19
337	容量规划 (CP)	18
340	更改管理 (CM)	
341	新产品推出与删除 (NPI/NPR)	
343	服务部署与退役 (SDD)	
345	性能与效率 (PE)	

其他运营职责

页次	运营职责	所在章节
347	服务交付：构建阶段	9
350	服务交付：部署阶段	10、11
351	减少“苦活”	12
353	灾难准备	15

A.1 常规任务

常规任务（RT）包括常规、非紧急运营任务的处理——如何接收、排队、分发、处理和验证工作，以及定期任务的安排和执行。所有服务都有某种常规、预定或者非预定的工作需要完成。Web 运营团队往往不执行直接客户支持，但是有团队间请求、来自利益相关方的请求以及从直接客户支持团队升级的问题。这些主题在第 12 章和第 14 章中介绍。

评估问题样板

- ☐ 常见的定期运营任务有哪些？
- ☐ 常见运营任务有没有剧本？
- ☐ 常规请求的 SLA 是什么？
- ☐ 如何识别新剧本条目的需求？谁编写新条目？谁编辑现有条目？
- ☐ 如何接收和跟踪用户请求？
- ☐ 常见用户请求有没有剧本？
- ☐ 用户请求不包含在剧本的情况发生的频度如何？
- ☐ 用户如何要求我们进行支持？（在线和物理位置）
- ☐ 用户如何知道怎样要求我们的支持？
- ☐ 用户如何知道哪些方面得到支持，哪些没有？
- ☐ 我们如何响应未支持事项的支持请求？
- ☐ 常规支持有何限制？（运营实践、远程还是现场）用户如何知道这些限制？
- ☐ 不同规模分类是否有不同处理？如何确定规模？
- ☐ 如果这个 OR 有企业标准做法，具体的做法是什么？这一服务如何遵循该做法？

第 1 级：初始

- ☐ 没有剧本，或者剧本已经过时、未被使用。

- ☐ 结果不一致。
- ☐ 不同人的做法不同。
- ☐ 两个用户的相同请求通常得到不同结果。
- ☐ 过程没有记入文档。
- ☐ 团队无法列举所执行的过程（甚至在较高的级别上）。
- ☐ 请求丢失或者无限期搁置。
- ☐ 组织无法预测正常任务完成所需的时间。
- ☐ 运营问题的报告不会引起注意。

第2级：可重复

- ☐ 团队支持的服务有一个有限的列表。
- ☐ 每个端到端过程都列举了每个步骤以及相互依赖性。
- ☐ 每个端到端过程各步骤处理都记入文档。
- ☐ 不同的人以相同方式完成任务。
- ☐ 很遗憾，流程中可以看到一些重复劳动。
- ☐ 很遗憾，多项任务需要的一些信息可能被每个需要它的步骤重复创建。

第3级：已定义

- ☐ 团队为大部分请求定义了 SLA，但是可能没有坚持。
- ☐ 每个步骤都有 QA 检查列表，在转移给下一步骤之前必须完成。
- ☐ 团队提前研究其他团队进行的过程更改。
- ☐ 多个步骤需要的信息或者处理只创建一次。
- ☐ 没有（或者极小）重复劳动。
- ☐ 启用新容量的能力是可重复过程。

第4级：已管理

- ☐ 定义的 SLA 得到计量。
- ☐ 所有步骤都有反馈机制。
- ☐ 缺陷和返工定期（每周？）审核。
- ☐ 事后剖析发布给所有人查看，报告草稿在 x 小时内可用，最终报告在 y 天内完成。
- ☐ 受影响团队定期审核警报。跨职能团队也定期审核警报。
- ☐ 过程更改请求需要数据以计量问题是否解决。
- ☐ 仪表盘用业务术语（不只是技术术语）报告数据。
- ☐ 每个“故障切换规程”都有“最后使用日期”仪表盘。

- ☐ 容量需求在需求出现之前预测。

第5级：优化

- ☐ 在过程更改之后，对比前后数据以确定成功。
- ☐ 如果前后数据说明没有改进，过程更改将被撤销。
- ☐ 已经付诸行动的过程更改有各种来源。
- ☐ 至少有一个过程更改来自每个步骤（在最近的历史中）。
- ☐ 周期时间逐月改进。
- ☐ 通过用提取的实际数据建立“如果……会怎样”的场景模型支持决策。

A.2 紧急响应

紧急响应（ER）包括运行中断和灾难的处理方式。这包括设计弹性系统以避免运行中断，以及在运行中断期间和之后执行的技术和非技术过程（响应与补救）。这些主题在第6、14和15章中介绍。

评估问题样板

- ☐ 如何检测运行中断？（自动监控？用户投诉？）
- ☐ 常见故障切换场景和运行中断相关任务有无剧本？
- ☐ 是否有值班日程表？
- ☐ 值班日程表如何建立？
- ☐ 系统是否能承受局部级别故障（组件故障）？
- ☐ 系统能否承受地理级别的故障（备用数据中心）？
- ☐ 运营人员是否在地理上分散（其他地区能否在较长的时期内相互覆盖）？
- ☐ 你是否编写事后剖析报告？事后剖析是否有必须完成的最后期限？
- ☐ 事后剖析是否有标准模板？
- ☐ 事后剖析是否审核，以确保行动项目完成？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第1级：初始

- ☐ 运行中断由用户而非监控系统报告。
- ☐ 没有人值班，总是由单一人员值班，或者每个人都在值班。
- ☐ 没有值班安排。
- ☐ 没有值班日程表。

- 对各种警报该做什么没有剧本。

第2级：可重复

- 监控系统联络值班人。
- 有值班表和升级计划。
- 有用于创建下月值班日程表的可重复过程。
- 任何可能的警报都存在剧本项目。
- 存在事后剖析模板。
- 事后剖析偶然编写，但是不一致。
- 值班覆盖范围在地理上分散（跨多个时区）。

第3级：已定义

- 运行中断根据规模分级（较小、严重、灾难性）。
- 已定义人们值班频率的限值（以及最小值）。
- 为所有重大运行中断编写事后剖析。
- 为警报响应定义 SLA：初始、人员到场、问题解决、事后剖析完成。

第4级：已管理

- 值班任务由最能解决问题的人分担。
- 根据策略验证人们的值班频率。
- 事后剖析经过审核。
- 有筛选事后剖析建议和确保它们完成的机制。
- 主动计量 SLA。

第5级：优化

- 频繁进行压力测试和故障切换测试（每季度或者每月）。
- “游戏日”演练（密集、系统级测试）定期进行。
- 监控系统在运行中断发生之前警报（指示“病态”系统而不是“停运”的系统）。
- 存在机制，人工激活任何最近没有使用的故障切换过程。
- 进行试验以改进 SLA 依从性。

A.3 监控与指标

监控与指标（MM）涵盖了收集和使用数据进行决策。监控收集有关系统的数据，指标使用数据计量性能的可量化部分。这包括带宽、速度或者延迟等技术指标；比率、总和、

平均值和百分位数等衍生指标；资源高效使用或者与服务水平协议（SLA）依从性等业务目标。这些主题在第 16、17 和 19 章中介绍。

评估问题样板

- ☐ 服务水平目标（SLO）是否记入文档？如何知道 SLO 与客户需求相匹配？
- ☐ 你是否有仪表盘？采用的是技术还是业务术语？
- ☐ 收集的数据和预测的精确性如何？如何得知？
- ☐ 服务的效率如何？机器是过度利用还是利用率不足？如何计量利用率？
- ☐ 如何计量延迟？
- ☐ 如何计量可用性？
- ☐ 如何知道监控系统本身是否停机？
- ☐ 如何得知用于计算关键绩效指标（KPI）的数据是否新鲜？是否有显示计量新鲜度和精确性的仪表盘？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第 1 级：初始

- ☐ 没有将任何 SLO 记入文档。
- ☐ 如果有监控，不是一切指标都得到监控，没有办法检查完整性。
- ☐ 系统和服务人工添加到监控系统（如果有的话）。没有任何过程。
- ☐ 没有仪表盘。
- ☐ 计量或者指标很少或者没有。
- ☐ 你认为客户很愉快，但是并非如此。
- ☐ 进行有利于某个个人或者小集团而不利于更大组织或者系统的优化是常见现象（并且得到奖赏）。
- ☐ 部门目标强调部门绩效，不利于组织绩效。

第 2 级：可重复

- ☐ 创建机器 / 服务器实例的过程，以确保监控它们。

第 3 级：已定义

- ☐ SLO 记入文档。
- ☐ 业务 KPI 已经定义。
- ☐ 业务 KPI 数据的新鲜度已定义。
- ☐ 存在一个系统，验证所有服务被监控。
- ☐ 监控系统本身受到监控（元监控）。

第4级：已管理

- ❑ SLO 记入文档并受到监控。
- ❑ 已定义 KPI 被计量。
- ❑ 有显示每个步骤完成时间的仪表盘，每步的滞后时间已确定。
- ❑ 存在显示当前瓶颈、积压工作和闲置步骤的仪表盘。
- ❑ 仪表盘显示缺陷和返工计数。
- ❑ 为监控系统和所有分析系统执行容量规划。
- ❑ 用于计算 KPI 的数据新鲜度受到计量。

第5级：优化

- ❑ 收集数据的精确性通过主动测试验证。
- ❑ KPI 使用 1 分钟以内的数据计算。
- ❑ 仪表盘和其他分析显示基于新鲜数据。
- ❑ 仪表盘和其他显示快速加载。
- ❑ 存储、CPU 和监控系统网络的容量规划以与任何重要服务相同的精密程度进行。

出乎意料的慢速缓存

Stack Exchange 购买了使用全球分布式缓存加速网页交付客户的产品。大部分客户部署了这个产品，并假定它有“不会丧失”的好处。

在部署之前，Stack Exchange 工程师 Nick Craver 创建了一个计量端到端页面加载时间的框架。目标是精确知道整体的改进程度以及对不同地理区域的客户的改进程度。

令人吃惊的是，他们发现该产品使性能降级。它改进某些方面但是却不利于其他方面，造成净性能损失。

Stack Exchange 和供应商一起识别问题。结果是，找到了一个重大设计问题并加以修复。

如果在更改前后没有认真计量性能，Stack Exchange 的努力就会在不知情的情况下使其服务变得更慢。令人疑惑的是，这一产品的许多其他客户都没有进行这种计量，而是简单地认定性能得到改进，而实际上性能却在恶化。

A.4 容量规划

容量规划（CP）涵盖了未来资源需求的确定。所有服务都需要为未来资源进行某种规划。容量规划包括技术方面——理解单位增长所需的资源数量，也包括非技术方面（如预算、预报和供应链管理）。这些主题在第 18 章中介绍。

评估问题样板

- ☐ 现在有多少容量?
- ☐ 预计从现在起3个月内需要多少容量? 12个月内呢?
- ☐ 使用何种统计模型确定未来需求?
- ☐ 如何测试负载?
- ☐ 容量规划需要花费多少时间? 可以采取什么措施简化?
- ☐ 指标是自动收集的吗?
- ☐ 可用指标是否始终可用? 是否需要初始化收集过程?
- ☐ 容量规划是无人承担、每个人都承担、特定人承担还是一个容量规划团队承担的工作?
- ☐ 如果对于这个OR有企业标准做法, 具体的做法是什么? 服务如何遵循这一做法?

第1级: 初始

- ☐ 没有保留任何库存。
- ☐ 系统常常耗尽容量。
- ☐ 确定添加多少容量是凭借传统、猜测或者运气。
- ☐ 运营部门的容量规划是被动的, 往往不能及时满足容量需求。
- ☐ 容量规划是每个人的工作, 因此无人负责。
- ☐ 没有专门的人员处理CP任务。
- ☐ 存在很大余量, 而不能精确地知道需要多少。

第2级: 可重复

- ☐ CP指标按照需要或者仅在需要时收集。
- ☐ 收集CP指标的过程是书面、可重复的。
- ☐ 负载测试偶然进行——可能在服务新推出时进行。
- ☐ 所有系统的库存情况是精确的, 可能是因为人工的努力。

第3级: 已定义

- ☐ CP指标自动收集。
- ☐ 增长所需的容量已做了很好的定义。
- ☐ 团队中有专门的CP人员。
- ☐ CP需求定义到子系统级别。
- ☐ 负载测试由重大软件和硬件更改触发。
- ☐ 库存作为容量变化的一部分更新。
- ☐ 已定义承受典型负载高峰所需的余量。

第4级：已管理

- ☐ CP 指标连续收集（每天 / 每周而不是每月或者每季度）。
- ☐ 附加容量在人工批准下自动获得。
- ☐ 测试期间检测性能衰退，如果性能衰退在生产环境中持续（不是缺陷），则涉及 CP。
- ☐ 仪表盘包含 CP 信息。
- ☐ 自动检测关联度的变化，发起 CP 单据验证和调整核心驱动力和资源单元之间的关系。
- ☐ 使用 MACD 指标或者类似技术自动检测意外增加的需求，为 CP 人员或者团队生成单据。
- ☐ 系统中的余量受到监控。

第5级：优化

- ☐ 对比过去的 CP 预测和实际结果。
- ☐ 负载测试作为持续测试环境的一部分进行。
- ☐ 团队雇佣统计人员。
- ☐ 附加容量自动获得。
- ☐ 余量进行系统性优化，减少浪费。

A.5 更改管理

更改管理（CM）涵盖一段时间内服务的有意更改。这包括软件交付平台——软件发行中涉及的步骤：开发、构建、测试和投产。对于硬件，这包括固件升级和小规模硬件调整。这些主题在第 9、10 和 11 章中介绍。

评估问题样板

- ☐ 部署（发行版本投产）的频率如何？
- ☐ 部署花费多少人工？
- ☐ 接收发行版本时，运营团队是否需要在投产前更改什么？
- ☐ 运营如何知道发行版本是主版本还是次版本，更改是大是小？这些类型的发行版本如何做不同处理？
- ☐ 运营部门如何知道发行是否成功？
- ☐ 发行失败的频率如何？
- ☐ 运营部门如何得知新版本可用？
- ☐ 是否有更改冻结窗口？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第 1 级：初始

- ☐ 部署审慎进行，因为它们的风险很大。
- ☐ 部署过程是临时且费工的。
- ☐ 开发人员在发行版本为部署做好准备时通知运营部门新的发行版本。
- ☐ 发行版本可用之后数周或者数月才部署。
- ☐ 运营和开发人员在部署发行版本时发生争吵。

第 2 级：可重复

- ☐ 部署不再是临时的。
- ☐ 部署人工进行但保持一致。
- ☐ 发行在交付时部署。
- ☐ 部署经常失败。

第 3 级：已定义

- ☐ 成功部署的组成部分已经定义。
- ☐ 次版本和主版本的处理不同。
- ☐ 发行版本可用和部署之间的预计时隙已定义。

第 4 级：已管理

- ☐ 部署成败根据定义计量。
- ☐ 部署很少失败。
- ☐ 发行版本可用和部署之间的预计时隙受到计量。

第 5 级：优化

- ☐ 采用持续部署。
- ☐ 部署极少失败。
- ☐ 新发行版本部署延迟很小。

A.6 新产品推出与删除

新产品推出与删除（NPI/NPR）包含在某个环境中引入及删除新产品和服务。这是一个协调职能：新产品或者服务的推出需要可能触及多个团队的支持基础设施。

例如，在数据中心环境引入新型号的计算机硬件时，某些团队必须能够访问样板硬件，进行测试和资格认证，采购部门必须制定订购机器的过程，数据中心技术人员需要文档。为

了推出软件和服务，应该有需求收集、评估与采购、许可证和服务台及运营剧本创建等任务。

产品删除可能涉及找出所有安装特定旧操作系统版本的机器，查看它们是否升级。产品删除必须找出当前用户，就迁移的时间表达成一致，升级文档并最终退役产品、任何相关的许可证、维护合同、监控及剧本。大部分工作是团队之间的沟通和协调。

评估问题样板

- ☐ 如何在环境中引入新硬件？涉及哪些团队，如何沟通？过程花费的时间有多长？
- ☐ 如何从系统中淘汰旧硬件或者软件？
- ☐ 处置旧硬件采用何种过程？
- ☐ 在处置时采取什么步骤确保磁盘和其他存储删除？
- ☐ 新软件或者新服务如何引入？涉及哪些团队，如何沟通？过程花费的时间有多长？
- ☐ 团队之间采用何种移交过程？
- ☐ 使用哪些工具？
- ☐ 文档是否保持最新？
- ☐ 哪些步骤需要人工干预？如何消除这些步骤？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第 1 级：初始

- ☐ 新产品通过临时措施和个人英雄主义推出。
- ☐ 团队对 NPI 感到吃惊，往往发现自己必须在没有得到通知的情况部署某些产品投产。
- ☐ NPI 因为缺乏容量、沟通不良或者错误而推迟。
- ☐ 很少废弃旧产品，造成运营人员必须支持“无限”数量的硬件或者软件版本。

第 2 级：可重复

- ☐ NPI/NPR 所用过程是可重复的。
- ☐ 团队之间的移交是书面化的且达成一致。
- ☐ 每个团队都有与 NPR/NPI 相关的任务剧本。
- ☐ 设备擦除和处置都文档化并经过验证。

第 3 级：已定义

- ☐ NPI/NPR 预期花费时间已定义。
- ☐ 团队之间的移交编码为机器可读格式。
- ☐ 所有团队成员理解自己融入更大的整体过程中时的角色。
- ☐ 每个团队支持的最大产品数量已定义。

- ☐ 每个团队当前支持的产品列表可供所有团队使用。

第4级：已管理

- ☐ 有用于观察 NPI 和 NPR 进展的仪表盘。
- ☐ 主动修订和改进团队之间的移交过程。
- ☐ 跟踪不再支持的产品数量。
- ☐ 不再支持产品的退役是高优先级任务。

第5级：优化

- ☐ NPI/NPR 任务成为团队之间的一个 API 调用。
- ☐ NPI/NPR 过程称为团队负责的自助服务。
- ☐ 团队之间的移交是一个线性流程（或者对于非常复杂的系统，加入多个现行流程）。

A.7 服务部署与退役

服务部署与退役（SDD）涵盖了现有服务实例的创建和关闭（退役）。在服务设计之后，它通常被重复部署。部署可能包括在新数据中心中启用卫星副本，或者创建现有服务的一个开发环境。退役可能是关闭数据中心、缩减过剩容量或者关闭特定服务实例（如演示环境）等工作的一部分。

评估问题样板

- ☐ 启动一个服务实例采用何种过程？
- ☐ 关闭服务实例采用何种过程？
- ☐ 如何添加新容量？如何关闭未用容量？
- ☐ 哪些步骤涉及人工干预？如何消除这些步骤？
- ☐ 有多少团队涉及这些过程？
- ☐ 所有团队是否都知道如何融入全局？
- ☐ 团队之间的工作流程是什么样的？
- ☐ 使用哪些工具？
- ☐ 文档是否最新？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第1级：初始

- ☐ 过程没有记入文档，具有偶然性。结果不一致。
- ☐ 过程由“谁”做某事定义，而不是由所完成的工作定义。

- 请求因为沟通不畅、缺乏资源或者其他可避免的原因而延迟。
- 不同的人以不同方式完成任务。

第2级：可重复

- 部署或者退役服务所需过程已经理解并记入文档。
- 每个步骤的过程记入文档并检验。
- 每个步骤都有 QA 检查列表，必须完成才能移交给下一步。
- 团队预先学习其他团队对过程的更改。
- 多个步骤需要的信息或者处理只创建一次。
- 没有（或者最大限度减少）重复劳动。
- 启用新容量是可重复过程。
- 设备的擦除和处置记入文档并验证。

第3级：已定义

- 每一步所花费时间的 SLA 已定义。
- 对于物理部署，消除材料浪费（盒子、包装、容器）的标准基于本地环境标准。
- 对于物理退役，旧硬件处置标准基于本地环境标准，以及组织自己的数据擦除标准。
- 存在执行许多步骤和过程的工具。

第4级：已管理

- 为每个步骤定义的 SLA 受到计量。
- 所有步骤都有反馈机制。
- 缺陷和返工定期审核。
- 容量需求在需求出现之前预测。
- 设备处置依从性根据组织标准和本地环境法规计量。
- 计量部署中浪费的材料（盒子、包装、容器）。
- 计量设备处置数量。

第5级：优化

- 过程更改之后，对比前后数据以确定成功。
- 如果前后数据表明没有改进，过程更改被撤销。
- 过程更改有不同来源。
- 周期时间得到逐月改进。
- 决策通过提取真实数据建立“如果……会怎样”场景模型来支持。

- 设备处置通过设备部署缩减加以优化。

A.8 性能与效率

性能与效率 (PE) 包括资源使用的成本效益和服务的执行情况。运行中的服务必须有很好的性能且不浪费资源。我们通常使用更多资源改进性能, 也可能改进效率而不利于性能。两者兼得需要努力求得平衡。成本效益是资源成本除以使用数量。资源效率是资源数量除以使用数量。要计算这些统计数字, 必须知道存在多少资源, 因此需要某种库存管理。

评估问题样板

- 计量性能使用什么公式?
- 用什么公式确定利用率?
- 用什么公式确定资源效率?
- 用什么公式确定成本效益?
- 如何计量性能变化?
- 性能、利用率和资源效率是否自动监控? 这些计量是否都有仪表盘?
- 是否有这个服务使用的机器和服务器库存清单?
- 库存清单怎样保持最新?
- 如果库存清单中遗漏了某个设备, 如何得知?
- 如果对于这个 OR 有企业标准做法, 具体的做法是什么? 服务如何遵循这一做法?

第 1 级: 初始

- 性能和利用率没有一致的计量。
- 计量内容取决于安装系统和服务的人。
- 没有计量资源效率。
- 性能问题总是出其不意地来临, 因为数据不足, 难以诊断和解决。
- 库存清单不是最新的。
- 库存可能更新也可能没有更新, 取决于涉及项目接收和处置的人。

第 2 级: 可重复

- 通过所有系统和服务收集与性能和利用率相关的所有指标。
- 启动新系统和服务的过程已经记入文档, 每个人都遵循该过程。
- 系统在配置供服务使用时与服务关联, 在释放时解除关联。
- 库存清单最新。库存管理过程有很好的文档, 每个人都遵循该过程。

第3级：已定义

- ☐ 在安装过程中自动为所有系统和服务配置性能和利用率监控，在退役时删除。
- ☐ 每个服务的性能目标已定义。
- ☐ 每个服务的资源使用目标已定义。
- ☐ 面向服务的性能和利用率指标的公式已定义。
- ☐ 持续监控每个服务的性能。
- ☐ 持续监控每个服务的资源利用率。
- ☐ 监控任何服务当前没有使用的闲置容量。
- ☐ 预期余量已定义。
- ☐ 保持库存清单最新的任务和职责已定义。
- ☐ 已经部署跟踪与网络连接的设备及其硬件配置的系统。

第4级：已管理

- ☐ 仪表盘跟踪性能、利用率和资源效率。
- ☐ 跟踪余量的最小值、最大值和第90个百分位值，与预期余量对比，可见于仪表盘。
- ☐ 设定性能和效率目标并跟踪。
- ☐ 每个服务的性能和效率目标及状态定期审核。
- ☐ 使用KPI设定驱动最优化行为的性能、利用率和资源效率目标。
- ☐ 自动化系统跟踪网络上的设备及其配置，并将其与库存系统比较，标记找出的问题。

第5级：优化

- ☐ 用性能仪表盘识别瓶颈，结果是进行更改。
- ☐ 识别使用大量资源的服务并进行更改。
- ☐ 如果更改没有正面效果则撤销。
- ☐ 定期评估计算机硬件型号，找出不同资源的利用率更为平衡的型号。
- ☐ 定期评估硬件的其他来源和其他硬件型号，以确定成本效益能否改进。

A.9 服务交付：构建阶段

服务交付是创建服务的技术过程，它从开发人员创建的源代码开始，以在生产环境中运行的服务结束。

评估问题样板

- ☐ 软件如何从源代码构建为软件包？

- ☐ 最终软件包是从源代码构建，还是由开发人员交付预先编译的元素？
- ☐ 单元测试的代码覆盖率是多少？
- ☐ 哪些测试是全自动的？
- ☐ 是否收集关于缺陷提前期、代码提前期和补丁提前期的指标？
- ☐ 为了构建软件，所有原始源文件是否都来自版本控制存储库？
- ☐ 为了构建软件，访问多少个地方（存储库或者其他来源）以获得所有原始源文件？
- ☐ 结果软件是以包还是一组文件的形式交付？
- ☐ 部署所需的一切要素是否都在包中交付？
- ☐ 使用哪个包存储库将结果移交给部署阶段？
- ☐ 是否有用于所有步骤状态与控制的单一构建控制面板？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第 1 级：初始级

- ☐ 每个人都在自己的环境中构建。
- ☐ 人们签入代码而没有检查构建的情况。
- ☐ 开发人员交付预先编译的元素供打包。
- ☐ 很少（或者没有）执行单元测试。
- ☐ 没有收集任何指标。
- ☐ 没有使用版本控制系统存储源文件。
- ☐ 软件构建是一个人工过程，或者具有人工步骤。
- ☐ 有些源文件的主拷贝保存在个人主目录或者计算机上。

第 2 级：可重复

- ☐ 构建环境已定义。每个人使用相同的系统，获得一致的结果。
- ☐ 软件构建仍然是人工完成的。
- ☐ 测试人工进行。
- ☐ 存在一些单元测试。
- ☐ 源文件保存在版本控制的存储库中。
- ☐ 软件包作为交付最终结果的手段。
- ☐ 如果支持多个平台，每个平台的过程都是可重复的，但是可能是单独进行的。

第 3 级：已定义

- ☐ 软件构建是自动化的。
- ☐ 自动化构建的触发器已定义。
- ☐ 单元测试覆盖率预期已定义：低于 100%。

- ❑ 缺陷提前期、代码提前期和补丁提前期的指标已定义。
- ❑ 每一步的输入和输出已定义。

第4级：已管理

- ❑ 计量构建成功/失败比率并在仪表盘上跟踪。
- ❑ 自动收集缺陷提前期、代码提前期和补丁提前期的指标。
- ❑ 指标在仪表盘上显示。
- ❑ 计量和跟踪单元测试的覆盖率。

第5级：优化

- ❑ 使用指标选择优化项目。
- ❑ 尝试优化涉及收集前后指标的过程。
- ❑ 每个开发人员可以在自己的沙箱中执行端到端的构建过程，然后将更改提交到中心存储库。
- ❑ 不充分的单元测试代码覆盖会阻止投产。
- ❑ 如果支持多个平台，其中一个的构建和所有其他平台上的构建一样简单。
- ❑ 软件交付平台用于构建基础设施，也用于构建应用程序。

A.10 服务交付：部署阶段

部署阶段的目标是创建一个运行环境。部署阶段在一个或者多个测试及生产环境中创建服务。然后，这些环境用于测试或者实际生产服务。

评估问题样板

- ❑ 软件包如何在生产环境中部署？
- ❑ 在生产环境中部署服务需要多长的停机时间？
- ❑ 是否收集关于部署频率、服务平均恢复时间和更改成功率的指标？
- ❑ 将软件包从测试升级到生产的决策是如何做出的？
- ❑ 完成哪些类型的测试（系统、性能、负载、用户验收）？
- ❑ 如何为小、中、大型发行以不同方式处理部署？
- ❑ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第1级：初始

- ❑ 部署涉及或者需要人工步骤。
- ❑ 测试和生产环境的部署是不同的过程，每个都使用自己的工具和规程。

- ☐ 团队中的不同人以不同方式执行部署。
- ☐ 部署需要停机，停机的时间有时很长。
- ☐ 发行版本升级到生产是临时的，或者定义不清晰。
- ☐ 测试是人工、定义不清晰的，或者完全没有进行。

第 2 级：可重复

- ☐ 部署在文档化、可重复的过程中进行。
- ☐ 如果部署需要停机时间，这个时间是可以预测的。
- ☐ 测试过程文档化、可重复。

第 3 级：已定义

- ☐ 部署频率、服务平均恢复时间和更改成功率指标已定义。
- ☐ 部署引起停机时间的计量已定义。限制和预期已定义。
- ☐ 发行版本升级到生产环境的过程已定义。
- ☐ 测试结果清晰地传达给所有利益相关方。

第 4 级：已管理

- ☐ 自动收集部署频率、服务平均恢复时间和更改成功率的指标。
- ☐ 指标在仪表盘上显示。
- ☐ 部署引起的停机时间自动计量。
- ☐ 计量部署期间减少的生产容量。
- ☐ 测试是全自动的。

第 5 级：优化

- ☐ 使用指标选择优化项目。
- ☐ 尝试优化涉及收集前后指标的过程。
- ☐ 部署是全自动的。
- ☐ 升级决策完全自动化，可能有少数特定的例外。
- ☐ 部署不需要停机时间。

A.11 减少“苦活”

减少“苦活”是改进系统中人员利用水平的过程。当我们减少苦活（令人疲劳的体力劳动）时，就为运营人员创造了更可持续的工作环境。尽管减少“苦活”本身不是一个服务，这个 OR 可用于评估苦活的数量，确定采用何种方法加以限制。

评估问题样板

- ☐ 每周有几个小时花在编码上，几个小时花在非编码项目上？
- ☐ 花在项目工作上和可以自动化的手工劳动上的时间各占多大比例？
- ☐ 花在手工劳动上的时间比率达到多少会触发警报？
- ☐ 采用什么过程检测手工劳动比率超出危险阈值？
- ☐ 采用什么过程提出危险警报？谁负责？
- ☐ 提出警报之后发生什么情况？何时降低手工劳动的比例？
- ☐ 如何识别减少苦活的项目？如何排定其优先级？
- ☐ 这些项目的效能如何计量？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第 1 级：初始

- ☐ 苦活没有得到计量，不断增长直到任何项目工作（或者几乎任何项目工作）都无法完成。
- ☐ 没有发出危险信号的过程。
- ☐ 有些人发现苦活成为问题并寻求解决方案，但是其他人没有意识到这个问题。
- ☐ 人们选择对自己最有趣的项目，没有注意哪些项目有最大的影响。

第 2 级：可重复

- ☐ 花在苦活和项目上的时间已得到计量。
- ☐ 定义花在苦活上的时间比例阈值并传达。
- ☐ 发出危险信号的过程文档化并传达。
- ☐ 每个人跟踪自己的苦活和项目工作的比率，并负责发出危险信号。
- ☐ 危险信号不总是在应有的时候发出。
- ☐ 对苦活减少有最大作用的项目识别过程已定义。
- ☐ 排定项目优先级的方法已经文档化。

第 3 级：已定义

- ☐ 对于每个团队，确定负责跟踪苦活和发出危险信号的个人。
- ☐ 涉及苦活减少项目识别和优先级排定的人已知。
- ☐ 苦活的危险水平和目标水平已定义。当苦活达到目标水平，降低危险水平。
- ☐ 在危险信号出现期间，团队只进行影响最大的苦活减少项目。
- ☐ 在危险信号出现期间，团队得到管理层的支持，搁置其他项目直到将苦活减少到目标水平。
- ☐ 在项目中的每一步之后，密切监控关于苦活的统计指标，提供有关正面或者负面变

化的反馈。

第4级：已管理

- ☐ 项目时间和苦活在仪表盘上跟踪，花在每个单独项目或者人工任务上的时间也得到跟踪。
- ☐ 危险信号自动发出，仪表盘给出了问题所在位置的概况。
- ☐ 监控时间跟踪数据以了解趋势，为一个或者多个领域中苦活增加的团队及早提供警报。
- ☐ 定义和跟踪 KPI，将苦活保持在预期范围中，并最大限度地缩短危险信号期间。

第5级：优化

- ☐ 调整目标和危险信号级别，监控结果对整体流程、绩效和创新的影响。
- ☐ 引入对主项目优先级排定过程的更改，并评估其正面或者负面的影响，包括对苦活的影响。
- ☐ 引入和评估对危险信号苦活减少任务优先级排定过程的更改。

A.12 灾难准备

运营组织必须能够很好地处置运行中断，必须有能够降低重复过去错误可能性的方法。灾难和重大运行中断确实会发生，公司上下的每个人必须承认这一事实，采取接受运行中断并从中学习的心态。系统应该设计为对故障具有弹性。

评估问题样板

- ☐ 什么是 SLA？采用什么工具和过程确保符合 SLA？
- ☐ 剧本的完整性如何？
- ☐ 剧本中的各个场景最后一次演练是什么时候？
- ☐ 采用何种机制演练不同的故障模式？
- ☐ 新团队成员如何培训，为处理灾难做准备？
- ☐ 在灾难发生期间适用哪些角色和职责？
- ☐ 如何为灾难做准备？
- ☐ 灾难如何用于改进未来的运营和灾难响应？
- ☐ 如果对于这个 OR 有企业标准做法，具体的做法是什么？服务如何遵循这一做法？

第1级：初始

- ☐ 灾难以临时的方式处理，需要个人英雄主义。

- ☐ 不存在剧本，或者没有覆盖所有场景。
- ☐ 很少或者没有培训。
- ☐ 从不测试服务弹性和不同的故障场景。

第2级：可重复

- ☐ 对所有故障模式（包括大规模灾难）都有剧本。
- ☐ 新团队成员接受在职培训。
- ☐ 灾难的处理是一致的，与响应者无关。
- ☐ 如果多个团队成员响应，其角色、职责和移交没有清晰定义，导致一些重复劳动。

第3级：已定义

- ☐ SLA 已定义，包括事后剖析报告的日期。
- ☐ 移交规程已定义，包括需要执行和记录文档的检查。
- ☐ 扩展响应团队以有效利用更多团队成员的方法已定义。
- ☐ 团队成员在灾难中的角色和职责已定义。
- ☐ 定义并执行新团队成员的特殊灾难准备培训。
- ☐ 团队定期进行灾难准备演练。
- ☐ 演练包括在实际服务上进行的应急演练。
- ☐ 在每次灾难之后都生成事后剖析报告并传阅。

第4级：已管理

- ☐ 用仪表盘跟踪 SLA。
- ☐ 过程中从事件发生起每一步的时间都在仪表盘上跟踪。
- ☐ 灾难准备培训的计划确保覆盖所有方面。
- ☐ 灾难准备计划计量灾难准备培训的结果。
- ☐ 团队在灾难处理上变得更好，培训扩展为覆盖更复杂的场景。
- ☐ 各个团队都参与多个团队及服务的跨职能应急演练。
- ☐ 根据 SLA 跟踪和计量发布初始和最终事后剖析报告的日期。

第5级：优化

- ☐ 从仪表盘确定改善的领域。
- ☐ 测试新技术和过程，计量结果用于进一步的决策。
- ☐ 自动化系统通过人为导致故障（如果故障没有自然发生），确保每个故障模式在一定时期内演练。

分布式计算和云的起源及未来

对我来说很明显，这只是一小段历史的重现。

—Propellerheads

现代化的大型数据中心通常由许多个如比萨盒般大小的计算机的机架组成。这是任何大型云规模分布式计算环境的典型设计模式。为什么会采用这种方式？这么多公司是如何采用相同设计模式的呢？

答案是，这种设计模式是无法避免的。为了管理庞大的数据中心，硬件和过程必须有条不紊。在数据中心内部重复相同的机架结构可以降低复杂性，减少管理开销。许多其他因素导致选择适合这种机架的硬件。

不过，没有什么真正是无法避免的。因此，我们说充满标准硬件组合而成的机器的大型数据中心的出现、分布式计算代替大型计算机以及云计算的流行是不可避免的，只是因为需求是创造之母。

实现大型电子商务网站的早期方法并没有采用正确的规模经济。实际上，当时的扩展会带来更高的单位用户成本。这导致第一次 .com 泡沫无法持续。推动分布式计算革新的是创建正确规模经济的需求，在这种经济模式下，支持的用户越多就越经济。

计算成本每改善一个数量级，就会开创一个应用的新时代，每个时代的成就都是几年前所无法想象的。每个时代都需要新的支持基础设施技术以及运营方法论。这些技术和方法论不总是及时出现的，有时候它们会先于所在的时代。

理解这些变化的历史背景，就能为本书描述的工具和技术提供一个语境。如果你对此不感兴趣，可以跳过本附录。

计算机行业的历史相当漫长，可以上溯到算盘的出现。本附录将跳过一段岁月，专注

于几个特定的时期：

- Web 前时代：Web 出现之前的几年，1985～1994 年。
- 第一个 Web 时代：泡沫，1995～2000 年。
- .com 炸弹时代：2000～2003 年的经济低迷时期。
- 第二个 Web 时代：2003～2010 年的复苏。
- 云计算时代：我们今天所处的时代。

每个时期都有不同的需求，推动同时代的技术决策，逐步推动革新。这些需求由于经济大潮的起落、科技进步的进程和对可靠性和速度日益提高的预期而不断变化。本附录提供我们对各个时期发生的事件的解读。

B.1 Web 前时代（1985—1994）

在 Web 之前的计算和现在不同。可靠性和伸缩性很重要，但是不像今天的要求这么高。互联网只掌握在少数技术人员手中，大部分人没有意识到它的存在。

可用性需求

对于大部分企业来说，他们的内部运营依赖于计算机，但是运行中断大都对外部客户不可见。客户占人口总数相当大的部分、从企业场所之外访问服务的公司是个例外——例如，为客户提供电话呼叫服务的电话公司和提供 ATM 服务的银行。客户希望这些服务是 24×7 可用的。

在这个时代，大部分大型企业已经在大量工作上严重依赖计算机了。有些雇员从家中通过基于电话的调制解调器进行远程访问，有时候使用哑终端，有时则使用 PC 或者 Mac。

大部分企业可能安排停机维护——通过关注产品发行安排、避免季末时期。计算机系统的客户是公司内部人员，联系他们安排停机时间有很简便、已定义的方法。

互联网大体上是基于文本的，有电子邮件、互联网新闻、电子公告牌和文件传输程序。互联网服务的运行中断可能造成电子邮件或者新闻的积压，但是大部分人可能不会注意到。对于大部分人来说，互联网访问是一种额外的服务，对业务并不重要。有些公司为第三方（如需要支持的客户）提供匿名的 FTP 收件箱。其他商业化的业务很少在互联网上进行。在互联网的早期，将互联网用于商业化目的是否可能违反可接受的使用政策尚不清楚。

在这一时代，24×7 运营对于某些内部企业系统是很重要的，但对互联网服务就不是这样的。内部系统的维护性停机可以计划，因为用户数是已知的，通知起来也很容易。

停机时间过去是正常的

Tom 的第一次计算机体验是在 1980 年。Tom 当时正在上 6 年级，学校让一小组学生访问连接到主机的一个终端。当有人登录到系统时，它会显示 “the system will be down

every day after 6 PM for taping”（系统将在每天晚上 6 点之后停机进行磁带备份）。Tom 和他的老师都不知道这是什么意思，几年之后 Tom 意识到“taping”指的是“用磁带备份”。每天这个系统都要停机数小时进行备份，这是正常现象。

技术

商业应用和服务运行于主机和小型机上。这些设备往往有数兆字节存储，每秒可执行不到 100 万条指令（MIPS）。它们是服务器级别的机器，具有高质量的组件、高速度的技术和扩展能力。

家用计算机是新事物。不能访问互联网，常见的应用（独立游戏、字处理、电子表格和税收软件包）保存在软盘上。硬件组件是“消费级”的，意味着便宜、性能低下、不可靠。这些计算机常常依赖 8 位和后来的 16 位处理器，RAM 和磁盘存储空间以 KB 和 MB 计算，而不是现在的 GB 和 TB。CPU 的速度以 MHz 计算，GHz 是科幻小说中的事情。数据保存在速度缓慢的软盘上。10 MB 硬盘是奢侈品，而且很脆弱，跌落就可能将其摧毁。

伸缩性

公司的计算机系统服务于内部客户、应用程序和业务过程，必须随着公司的成长而扩展。随着公司业务的增长，员工数量和计算机的需求都会增长。即使对于快速增长的公司，这种增长也是相对容易预测的，往往受到低得多的新员工雇佣速度的制约。

业务关键应用运行于少数大型高端计算机上。如果计算机系统的容量耗尽，可以用更多的磁盘、内存和 CPU 升级。进一步的升级意味着购买更大的机器，因此当时的服务器通常在购买时有丰富的备用升级容量。有时候，服务还通过在多个地理位置（或者业务单位）部署应用服务器进行扩展，每个位置使用本地的服务器。例如，当 Tom 最初在 AT&T 工作时，公司的每个分部都有不同的工资处理中心。

高可用性

有高可用性需求的应用需要“容错”计算机。这些计算机有多个 CPU、错误校正 RAM 和其他当时极其昂贵的技术。容错系统是定制产品，通常只有军方和华尔街需要这类系统。因此，它们的价格通常超出了一般公司的承受能力。

成本

在这个时代，互联网对业务并不重要，内部业务关键系统可以安排停机，因为客户基础是一组已知的有限人员。在必要时，可以用很昂贵的专用硬件处理高级的可靠性及伸缩性需求。但是，企业很容易计算任何运行中断的成本，进行风险分析以理解提供更高可靠性的预算，在如何处理上做出明智的决策。这些成本都在控制范围之内。

伸缩性通过硬件升级处理,但是计算需求的扩展可以从公司的业务中预测。随着业务的增长,计算基础设施的预算增长,可以规划和安排升级。

B.2 第一个 Web 时代: 泡沫 (1995—2000)

在第一个 Web 时代开始时,网站是相对静态的链接文档库。第一批企业网站大体都是营销文献——关于公司及产品的信息、新闻报道、职位列表和各种面向客户组织的联络信息,如销售和客户服务。但是许多企业很快就意识到在互联网上开展业务的潜力,电子商务诞生了。

可用性需求

对于大部分公司来说,最初互联网存在和关键内部系统一样处理。它们使用拥有一些高可用性选项的服务器级硬件和附加的容量来实现伸缩性。大部分公司的互联网存在只是基础设施的另一部分,而不是业务中特别关键的部分。在正常条件下,网站应该保持正常运行,但是系统管理员可以按照需要计划停机维护。Web 服务的计划停机通常包含配置另一台机器,用一个静态页面响应,让人们知道网站停机维护,在稍后再次重试。

之后出现了一些新的创业公司,其业务模型是完全在 Web 上开展业务。这些公司没有现有的产品、渠道和客户,它们不采用包含 Web 的现有业务过程。对于这些公司,互联网存在就是整个企业。如果它们的 Web 销售渠道失效,一切都停顿了。它们无法联络客户以安排窗口维护,因为无从得知在这个时期内有哪些客户。任何能够访问互联网的人都是潜在客户。这些公司需要高可靠性的互联网存在,这种要求是前所未有的,这类企业需要 24×7 的运营,没有维护窗口。

技术

在这一时代,家用电脑变得更加常见,采用 xDSL 和有线电视公司提供的互联网服务等更快连接的家庭也更为普遍了。更好的图形和更强大的电脑催生了更好的游戏,其中许多都是联网的多用户游戏。IP 语音 (Voice over IP, VoIP) 出现,带来了相关的新产品与服务。磁盘变得更大、更便宜,人们开始数字化和存储新型数据,如音乐和视频。各大公司不可避免地围绕这些数据构建产品和互联网服务。

在服务器端,各大公司试图从购买 RAID 子系统 (代替普通磁盘)、多个高端处理器等入手,提供更可靠的互联网存在。虽然这些技术现在已经很常见,但是那时候还很昂贵。之前向需要大型高可靠性计算机的很小一部分客户销售的供应商,现在有了一组全新的客户,这些客户也希望机器既大又可靠。

Sun 微系统公司的 E4500 服务器是 .com 时代的标准硬件。这一时代的市场上还出现了负载平衡器。负载平衡器处于一组提供相同服务的机器之前,持续测试每台机器上的服务

是否可用。负载均衡器可以配置为主/从模式——将所有流量发送给主设备直到其出现故障，然后自动切换到从设备。它们还可以配置为在所有提供相同服务的机器之间分担负载。在后一种模式中，当一台机器停止正确响应，负载均衡器停止将查询导向该机器，直到它恢复正确响应。负载均衡器在机器停机时提供自动故障切换，成了需要高可用性的服务的实用工具。

伸缩性

企业需要服务器来为网站提供动力。系统管理员对新的需求应用旧方法：每个网站一台机器。随着网站越来越流行，为了满足需要而使用更大的机器。为了实现更好的性能，开发出了定制 CPU 和新的内部架构，但是这些机器很昂贵。软件也很昂贵。典型的 Web 服务器需要 OS 许可证、Web 服务器许可证和数据库服务器许可证。每个许可证的价格都和硬件成正比。

Web 对伸缩性的需求不仅受限于公司员工数量。Web 引入了一个环境，服务的用户可以是具备互联网连接的任何人。这是非常庞大且快速增长的数字。当网站推出或者成功地营销一个基于 Web 的服务，访问服务器的人数可能在短期内极快地增加。为突然、不可预测的服务使用率激增而进行扩展是一个新的挑战，在这一时代尚无法很好地理解，但是这是互联网创业公司试图准备的条件。互联网创业公司为取得成功而做出规划，购买可以买到的最大、最可靠的系统。这是一个昂贵的时代。

但是，通过更大、更可靠、更昂贵的硬件实现网站增长的方法在经济上是不可行的。一般来说，随着公司的成长，规模经济能够带来更低的单位成本。但是 .com 公司却随着增长而需要单位容量成本更高的计算机。性能超过普通计算机 10 倍的计算机，其成本超过 10 倍。这些更大、更可靠的系统使用定制硬件，市场更小——这两个因素推升了价格。为了在性能上得到线性增长，单位容量成本呈超线性增长。网站的用户越多，提供服务就越昂贵。这种模型违背了人们的期望。用于确保高可用性的技术需要额外的成本，下一节将对此进行介绍。

奇怪的是，当时这种高成本却为人们所接受。.com 公司的现金流很充足，在昂贵的机器上投入金钱是普遍现象，因为这表现了对公司成功的乐观。而且对网站规模化经济的理解尚不充分——增加销售量得到较低单位成本的传统模型仍然被认为是正确的。此外，那个时代创业公司的估值方式相当奇怪。如果创业公司实现利润很少，估值就很低。但是，如果它出现亏损，反而得到了高估值。而且亏损越大，估值越高。尽管这些创业公司的商业模式毫无意义，但是当它们出现在股票市场时，其股价一飞冲天，投资者获得大量利润。

当时有种说法：“我们可能在每次销售中损失金钱，但是将从总量上赚回来。”这种说法的背后是公司如果能够垄断市场、以后就可以提高价格的想法。但是，这种假设没有考虑到下一家创业公司可能立刻用自己的“以亏损换取垄断地位”方案打破高价格的情况。

“淘金热”的心态使这种经济现象流行了一阵子。2000 年泡沫破裂了。“纸牌屋”崩塌。

这种扩展基于 Web 服务的方法无法持续。即使泡沫没有因为投资问题和现金流而破裂，也会因为使用这种糟糕的技术经济学而破裂。

高可用性

随着 Web 的出现，用户基础从可预测、循环可用性（工作日朝九晚五）需求及访问日程的已知内部公司员工，变为需要不间断访问的未知互联网用户。这一变化造成了更高可靠性的需求。满足这种可用性目标的第一种方法是购买具有内建高可靠性的更昂贵硬件，例如 RAID 和多 CPU。但是即使是最可靠的系统也会偶尔出现故障。传统的关键系统选项是确定 4 小时更换配件服务合同，或者购买备件保存在机器附近。任何一种方法在硬件故障时都需要一定的停机时间。

还有一个问题，就是需要停机时间执行软件升级。应用内部企业通知用户的方法，网站预先公布停机时间成了普遍的做法。这时在网站上会出现“本网站将在太平洋标准时间周六正午到下午五时停机升级”之类的警告。普通的用户将会据此做出计划，但是新客户或者偶然登录的客户没有意识到这一点，因为没有任何途径可以通知他们。预先公告升级还可能在升级情况不佳的时候造成负面的新闻报道，因为有些人会注意升级的情况，并发表关于新服务的报道。

N+1 配置

使用两台机器成了最佳实践。一台机器运行服务，另外一台闲置，但是配置为在第一台机器出现故障时接管。除非网站有负载均衡器，否则“故障切换”通常需要人工干预，但是好的系统管理员可以快速切换，使网站的停机时间小于 1 个小时。这称作 N+1 配置。因为有一台超出需要的设备提供服务。考虑到任何时候都有 50% 的投资闲置，这种技术非常昂贵。

软件升级可以通过在推出新功能时升级并切换到备用服务器来完成。这样，停机时间可能只需要几分钟或者几秒钟。用户甚至可能没有注意到！

N+2 配置

如果备用服务器升级的时候主服务器出现故障怎么办？半配置的机器不处于可用状态。随着软件发行频率的提高，备用服务器处于不可用状态的可能性也在提高。

因此，最佳实践变成有 3 台机器——N+2 配置。现在系统管理员可以安全地执行升级，但是任何时候都有 66% 的硬件投资闲置。想象一下买了 3 座房屋却只在一处居住的情况。想象一下如果你是那个必须告诉 CEO 在闲置设备上花了多少钱的人……

有些公司试图通过机器间的负载共享进行优化。这需要额外的软件开发或者负载均衡器，但这是一个可行的方案。在 N+1 配置中，系统管理员可以将一台机器退出服务进行软件升级，其他机器保持运行。但是，如果两台机器的利用率都达到 80%，这时单一机器的

利用率将达到 160%，最终用户可能无法接受十分缓慢的速度，网站可能因此停运。对这个问题最简单的解决方案是绝不让任何一台机器的利用率超过 50%——但是这仍然使我们处于为半数闲置容量买单的境地。只是，闲置容量分到两台机器上！

有些公司试图只在深夜进行这种升级，此时用户较少，利用率会降到 50% 以下。这种方法使得可进行升级的时间很少，大型、复杂的升级变得极其危险。需要操作系统升级或者大规模测试的更改不可能完成。如果网站在国际上大受欢迎，在每个时区都很忙碌，这种选择就不存在了。而且，没有人能够安排硬件故障只发生在夜间！上述方法都不是能更好地利用可用资源的可行选项。

为了实现高可用性，网站仍然需要 3 台机器，造成 66% 的闲置容量。

成本

在这一时代，提供高可用性的流行 Web 服务的成本非常高。服务运行于昂贵的高端硬件上，使用 N+1 或者 N+2 配置的 RAID 和多 CPU 等昂贵可靠性功能。这种架构意味着这些昂贵的硬件中有 50–66% 是闲置的。为了将停机时间降低到接近 0，网站可能使用昂贵的负载均衡器。

这种高端硬件的 OS 成本和支持成本也很高，应用软件（如 Web 服务器和数据库软件）的成本也是如此。

扩展这样构建的网站意味着，随着网站的增大，所提高性能的单位成本将会提高。和更多客户造成的规模经济不同，客户越多意味着单位成本越高。这种成本模型和企业的正常预期相反。

B.3 .com 炸弹时代（2000—2003）

.com 泡沫的破裂始于 2000 年。到 2001 年，泡沫以尽可能快的速度缩小。大部分 .com 公司烧光了它们的风险投资，股票停止交易，这些公司往往从未盈利。

崩溃之后经历了一个平静期。发展稍微减慢。人们可以停下来考虑从过去学到的经验。没有了大肆宣传、营销和容易到手的钱，只有更好的技术存活下来。没有了因为投资者希望快速花钱、快速取得投资回报的压力，人们可以花费足够的时间思考和投资新的解决方案。硅谷的大部分人都是精英，有用的技术占据统治地位，大肆宣传和自我推销被过滤掉。

可用性需求

在 .com 炸弹时代，可用性需求没有明显的变化。从崩溃中存活下来的互联网公司可对可用性需求有了更好的理解，领会出在不倾家荡产的情况下满足这些需求的方法。

技术

3 个趋势推动了下一阶段的发展：之前的高景气时代留下的过剩容量、硬件的商品化以及开源软件的成熟。

第一个趋势是短命的，但是很重要。之前的高景气时代已经积累了许多容量，供应商正在杀价。地下和海中铺设了几百万英里光纤，以满足预测中全世界的带宽需求。由于客户相对少，电信供应商孤注一掷。类似地，巨型数据中心“主机托管设施”已经建立起来。主机托管设施是高可靠性的数据中心设施，向其他公司出租空间。许多主机托管供应商在建设世界上最大的设施之后破产，这些空间现在以很便宜的价格出租。虽然这些过剩容量最终会耗尽，但是暂时的低价格帮助推动了这个时代。

第二个趋势是家用计算机中使用的硬件组件（如 Intel x86 CPU、低端硬盘和 RAM）的商品化。在 Web 出现之前，普通家庭没有计算机。互联网的流行带来了家用计算机的更大需求，造成这些组件以前所未有的规模制造。此外，需要高端图形设备、许多内存和快速 CPU 的游戏的流行，成了消费者市场上越来越高端设备制造的主要推动力。这种大规模制造导致商品化和更低的价格。家庭 PC 降价，但是服务器仍然使用不同的组件，保持高价格。

第三个趋势是 Linux、Apache、MySQL 和 Perl 等开源项目的成熟。Linux 的兴起将 UNIX 类服务器操作系统引入 Intel x86 平台。过去，使用 Intel x86 的系统无法运行服务器级的 UNIX 和 UNIX 类操作系统。SGI、IBM、Sun 和其他公司没有提供可用于 x86 的操作系统。Intel x86 计算机运行 Windows 95 及其变种，这些操作系统不是按照服务器操作系统设计的。即使设计为服务器操作系统的 Windows NT 也没有在 Web 服务平台上取得成功。

与此同时，还出现了可用于 x86 计算机的免费 BSD UNIX 版本。但是，最终 Linux 在 x86 UNIX 系统上占据统治地位，因为 RedHat 等多家公司提供带有支持的 Linux 版本。企业通常因为免费开源软件缺乏支持而回避它们。这些公司慢慢被说服，因为可以购买到支持，Linux 商业化版本对于服务器来说是可以接受的。尽管这些公司仍然为 OS 及其支持付费，但是它们意识到通过使用更便宜的 x86 硬件和 OS 成本的降低，可以显著地节约成本。

虽然 Linux 在第一个 Web 时代就已经出现，但是不够成熟，无法用于生产。实际上，和 Solaris OS、Netscape Web 服务器、Oracle 数据库、Java“栈”或者 Microsoft NT Server、IIS（Web 服务器）和 SQL Server 数据库及 .NET 环境相比，Linux、Apache、MySQL 和 Perl 等工具被视为玩具。尽管如此，这些开源项目现在创建了足以可靠地用于生产的软件。Linux 已经成熟，Apache 证明自己比商业化平台更快、更稳定、功能更丰富且更易于配置，MySQL 也比 Oracle 更容易安装和管理。Oracle 的价格很高，因此免费 SQL 数据库的出现水到渠成。Perl 也已成熟，加入了面向对象功能，在最初作为系统管理语言的小众市场之外获得了肯定。所有这些在几年之前还是无法想象的。开源软件的组合如此常见，因此产生了缩写词“LAMP”，意指 Linux、Apache、MySQL 和 Perl。使用商品化服务器运行免费操作系统的能力是革命性的。

高可用性

虽然 LAMP 系统比较便宜，但是它较慢，也较不可靠。要达到所替代的更大型机器的总处理容量，需要许多服务器。可靠性也成为更复杂的问题。

研究人员开始试验使用低成本组件建造服务器。加州大学伯克利分校（Patterson 等人，2002）的面向恢复计算（Recovery-Oriented Computing, ROC）等研究发现，许多小服务器可以运行的比单独的大服务器更好，也更便宜。这在以前是闻所未闻的。为桌面设计的低级 CPU 如何能与定制的 Sun SPARC 芯片抗衡？便宜的消费级硬盘如何能与昂贵的 SCSI 硬盘竞争？

工程上的常识是，组件越多意味着故障越多。如果一台计算机每 100 天可能出现一次故障，两台机组成的系统可能每 50 天出现一次故障。如果你有足够多的计算机，那么每天都可能遇到故障。

加州大学伯克利分校的 ROC 和其他研究项目颠覆了上述逻辑。在采用“分布式计算”时，计算机越多越可靠。这些研究人员开发了分布式计算技术，让许多机器分担工作负载并且有足够的备用容量，在一定比例的机器停机时不会影响服务，而不是设计一台计算机故障导致系统无法提供服务的系统。

ROC 还观察到，各个公司在每一级别上都为可靠性买单。负载均衡器提供可靠性，只向“正常运行”的服务器发送流量。服务器使用昂贵的定制 CPU 提供可靠性。存储系统使用昂贵的 RAID 控制器和高端硬盘提供可靠性。但是，为了避免由于需要人工故障切换的应用软件而造成的停机时间，各个公司仍然必须编写软件以便在运行中断时存活下来。为什么为这么多可靠性层次付费之后还要花钱开发能在运行中断中存活的软件？ROC 得出结论，如果软件已经能够在运行中断中存活，为什么还要为每个级别的额外质量付费？使用便宜的商品化组件制造的服务器可能可靠性较差，但是软件技术可以带来整体更可靠的系统。因为无论如何都必须开发软件，这样做更有意义。

数字化电子设备要么工作，要么不工作。提供服务的能力也是二元的：计算机开或者关。这被称作“运行、运行、运行、死亡”问题。

数字化电子设备通常一次又一次地运行，直到出现故障。在那个时候，它们完全失效，系统停止工作。这和旧电子管收音机等模拟系统形成对比。新的模拟系统工作得很好，随着时间的推移，零件开始老化，音质下降。用户听到更多静电干扰声，但是收音机仍然可以使用。静电干扰慢慢增加，在收音机无法使用之前数月一直警告着用户。模拟系统不会“运行、运行、运行、死亡”，而是慢慢降级。

利用分布式计算技术，每台单独的机器仍然是数字化的：它要么运行，要么不运行。但是，整组机器更像模拟的收音机：随着越来越多的机器失效，系统仍然能工作，但是性能下降。单一机器停机不会发出警报，而是在太多机器也出现故障之前必须进行维修的信号。

伸缩性

ROC 研究人员阐述了分布式计算足以可靠地提供需要高可用性的服务。但是这样的系统是否足够快呢？这个问题的答案也是肯定的。用足够多的基于商品化硬件、比萨盒大小的机器，可以实现满载的 Sun E10K 的计算能力。

Web 应用程序特别适合于分布式计算。想象一种简单的情况——网站的内容可以保存在一台机器上。大型服务器可以提供 4000 次查询 / 秒 (QPS) 的服务。假定一台商品化服务器只能提供 100QPS，花费 40 台服务器就可以得到较大机器的总容量。分布式计算的负载均衡算法也很容易扩展到 40 台机器。

数据伸缩性

对于某些应用程序，服务的所有数据可能不能只保存在一台商品化服务器上。这些商品化服务器太小，无法存储很大的数据集。Web 搜索等应用有一个相当大的数据集（“语料库”）。研究发现，通过将语料库分成许多“片段”，每个片段在不同机器（“叶子”）上存储，可以解决这个问题。其他机器（“根”）接收请求，并将其转发到对应的叶子上。

为了使系统在故障时有更大的弹性，每个片段可以保存在两个不同的“叶子”机器上。如果有 10 个片段，可以有 20 个叶子。只要特定片段的两个叶子都正常运行，根将在它们之间分割流量。如果一个叶子失败，根将把与这一片段相关的所有请求发送给其余叶子。两个保存相同数据的叶子同时发生故障的可能性很小，即使发生，在替换算法将遗漏的数据加载到备用机器之前，用户也可能没有注意到他们的 Web 搜索返回的结果稍少一些。

伸缩性也可以通过复制实现。如果系统处理请求的速度不够快，可以通过增加叶子扩展。特定片段可以保存在 3 个或者更多地方。

算法随着时间的推移而变得更加复杂。例如，语料库不是分成 10 个片段，每个机器保存一个，而是分割为 100 个片段，每台机器保存 10 个。如果特定分片接受大量访问（它成为“热区”），该片段可以被放在更多机器上，而淘汰较不流行的片段。较好的算法可以造成布局更好、多样化、可动态更新的语料库数据。

适用性

这些算法特别适合于 Web 搜索和类似应用，在这些应用中，除了制作新语料库时整体替换之外，数据大部分呈静态（不更改）。相反，它们不适合于传统应用。毕竟，你不想要在机器故障时返回不完整结果的数据库上的工资系统。而且，这些系统缺乏传统数据库的与一致性和可用性相关的许多特性。

新的分布式计算算法一个又一个地实现新应用。例如，以大规模 Web 服务的形式提供电子邮件的渴望促成了更好的存储系统。随着时间的推移，更多的边缘情况被克服，分布式计算技术可以应用到更多的应用程序上。

成本

如果分布式计算可以更加可靠和快速,那么它是否能够更有成本效益?一台高可靠性的昂贵机器在成本上远远高于供同等容量的商品化机器。实际上,使用分布式计算,容量的单位成本往往便宜 3~50 倍。前面我们已经看到,使用 N+1 和 N+2 配置,大型服务器的 50%~66% 的容量是闲置的。假定需要一整个机架的机器提供等价的容量。一个机架通常容纳 40 台机器,加上两台备用机器(N+2 冗余)。现在,“浪费”的容量从 66% 下降到 5%。闲置容量的缩小使分布式计算设计获得了领先的优势。加上商品化市场的力量推动组件成本下降,进一步增进了这一优势。而且,购买许多机器还能得到批量的折扣——这在购买一两台大型机器时无法获得。

在 2003 年的文章《Web Search for a Planet: The Google Cluster Architecture》(整个星球的 Web 搜索: Google 群集架构)中,Barroso 等人写道:

使用基于 PC 的廉价群集和高端多处理器服务器相比有明显的成本优势,至少对于我们的高度可并行化应用程序来说是这样。价值 278 000 美元的示例机架包含了 176 个 2 GHz Xeon CPU、176 GB RAM 和 7 TB 磁盘空间。相比之下,典型的 x86 服务器包含 8 个 2 GHz Xeon CPU、64 GB RAM 和 8 T 磁盘空间,其价格大约为 758 000 美元。换言之,多处理器服务器的价格大约贵 3 倍,但是 CPU 少 22 倍,RAM 少 3 倍,仅在磁盘空间上稍微胜出。(Barroso、Dean & Hölzle, 2003)

将机器削减到仅包含必需的组件,还可以进一步缩减成本。这些机器不需要图形卡、声卡、音箱、键盘、USB 端口或者正面有漂亮图标的塑料挡板。即使减少一个项目只节约几美元,购买几千台机器累计起来也有明显的节约。有些公司(Yahoo!)和供应商一起建造符合其规格的定制计算机,而其他公司(Google)在成长起来之后从头开始设计自己的计算机。

所有这些变化改变了计算经济学。实际上,是它们成就了第二个 Web 时代。

在旧的经济条件下,规模越大,系统更昂贵且不成比例。而在新的经济条件下,规模越大,改善经济状况的机会越多。成本的增长不再呈超线性曲线,而是近于线性。

B.4 第二个 Web 时代(2003—2010)

在计算的历史上,成本改进的每一次飞跃都带来一波新的应用程序。分布式计算更好的经济学就是这样的飞跃。过去失败的、无法获得利润的 Web 服务,现在运营起来更经济,得到了新的机遇。

可用性需求

和以前一样,完全通过互联网开展业务的公司将目标瞄准全球的互联网用户。这意味着它们需要 24×7 的可用性,没有计划中的停机时间,尽可能没有计划外的停机时间。各

大公司开始在服务产品中应用新开发的分布式计算技术，以更有成本效益的方式满足这些可用性需求。

技术

许多公司接受了分布式计算技术。Google 采用这种技术提供比之前更快的 Web 搜索。以前，Web 搜索需要许多秒，有时候甚至几分钟或者几小时才能显示结果。Google 对自己不到半秒就能返回结果的能力感到骄傲，在每个页面的底部显示请求处理的毫秒数。

Google AdSense 等新型广告系统对每次点击收取几美分的费用，现在因为计算成本明显下降而变得有利可图。过去的商业模式在网站上销售广告需要很大的销售力量，才能让广告商购买网站上的广告。AdSense 和类似的系统是全自动的。潜在的广告商可以在规模巨大的在线广告位置拍卖中出价。这减少了（往往消除）销售力量的需求。公司所需要做的就是网站上添加一些 JavaScript 代码。很快，广告就开始出现，财源也滚滚而来。这种广告系统创造了新的商业模式，促进了全新行业的发展。

Web 托管变得更加经济，软件也更容易使用。这导致了“博客”（最初来自于“Web-log”一词）的发明，博客的运营所需的技术知识极少。不需要销售力量和大型的 IT 部门，个人也可以运营一个成功的博客。博主们可以将重点放在内容的创建上，广告网络会给他们寄去支票。

高可用性

利用分布式计算技术，高可用性和伸缩性变成紧密耦合的概念。用于确保高可用性的技术也有助于伸缩性，反之亦然。Google 公布了其发明的许多分布式计算技术。最早的一些技术包括 Google 文件系统（Google File System，GFS）和 MapReduce。

GFS 是扩展到几个 TB 数据的文件系统。文件以 64M 数据块的形式保存。每个块保存在 3 台或者更多商品化服务器中。数据访问通过直接与包含所需数据的机器通信实现，而不是通过可能成为瓶颈的中介机器。GFS 操作常常并行发生。例如，文件的复制不是一次一块地进行的。相反，保存源文件一部分的每台机器可以和其他机器配对，所有数据块并行传输。配置 GFS 保存每个数据块的 3 个以上副本，降低任何时刻文件的所有拷贝都处于一台故障机器上的可能性，增强了弹性。如果一台机器出现故障，GFS 主机将使用剩余的拷贝填充其他位置的数据，直到实现复制因子。提高复制水平还能改善性能。数据访问在所有副本中负载均衡。副本越多，总读取性能越高。

案例研究：MapReduce

MapReduce 是并行处理大批数据的系统。假定你需要处理 100TB 数据，从头开始读取这么多的数据需要很长的时间。作为替代，可以设置数十台（或者数百台）机器，每台并行处理数据的一部分。它们处理（Map）所读取的数据，创建一个中间结果。然后再处

理并归纳 (Reduce) 这些中间结果, 获得最终结果。MapReduce 完成分割数据并将工作转交给不同机器的困难工作。在软件开发者使用 MapReduce 时, 只需要编写 “Map” 函数和 “Reduce” 函数。其他的一切都由系统处理。

因为 MapReduce 是集中化的服务, 改进它时所有用户都将从变化中获益。例如, 在花费数天、涉及数千台机器的 MapReduce 运行中很有可能有一台机器出现故障。检测故障机器并将对应部分数据发送给另一台正常机器的逻辑包含在 MapReduce 系统中, 开发人员不需要操心故障。如果开发了更好的故障机器检测和替换方法, 这一改进将进入中央 MapReduce 系统, 所有用户都将得益。

这样的改善之一包括预测哪些机器将出现故障。如果一台机器处理所属部分数据时所花费的时间明显更长, 这种情况可能是硬盘开始失效或者其他问题出现的一个迹象。MapReduce 可以引导另一台机器处理同一部分数据。先完成处理的机器将 “获胜” 并保留其结果。另一台机器的处理将被放弃。这种抢占式冗余性可以节约数小时的处理时间, 特别是因为 MapReduce 的运行速度和最慢的机器相同。多年以来, 这类的优化已经开发了很多。

很快就出现了 MapReduce 的开源版本。现在, 其他公司很容易采用 MapReduce 技术。最流行的实现是 Hadoop, 它包含了一个类似于 GFS 的数据存储系统, 称作 HBase。

伸缩性

如果你花费金钱发明了创建高伸缩性、高可靠性分布式计算系统的方法, 那么提高投资回报率的最佳方法是该技术用在尽可能多的数据中心上。

一旦维护了数百个类似配置计算机的机架, 集中管理就变得很有意义。集中运营在经济上很有好处, 例如, 标准化使自动化运营任务成为可能, 从而减少所需的工作量。如果这些工作是人工进行的, 越大的系统需要的人工越多。当运营自动化时, 开发软件的成本将分摊到所有机器上, 每台机器的成本随着机器的增加而降低。

为了获得更高的伸缩效率, Google 和 Facebook 等公司更进一步。为什么不将数据中心看作一台巨型计算机呢? 计算机由通信 “总线” 连接内存、磁盘和 CPU。数据中心有一个网络, 就像连接计算单元和存储的总线。为什么不将数据中心当成一台大型的仓库级计算机设计和运营? Barroso、Clidaras 和 Hölzle (2013) 提出了 “仓库规模计算” 这个术语, 以概括将整个数据中心作为单一巨型计算机的想法。

成本

在这个时代, 各大公司还争相开发更好、更便宜的数据中心。开源运动降低了软件的成本。摩尔定理降低硬件的成本, 剩下的成本就是电力和运营本身。各大公司发现, 这些领域的改进将带来竞争优势。

在第二个 Web 时代之前，数据中心以难以置信的速度浪费电力。根据正常运行时间学会的说法，典型的数据中心平均电力使用效率（Power Usage Effectiveness, PUE）为 2.5，这意味着，电表中的每 2.5 瓦电能中只有 1 瓦用于 IT 任务。通过使用最高效的设备和最佳实践，大部分设施可以实现 1.6 的 PUE。缺乏效率的地方主要有两个。首先，每当进行电能转换时效率都会有所降低。UPS 将电源从交流电（A/C）转换为直流电（D/C），然后又转换回 A/C——两次转换。电能从高压线转换为电源插座中的 110 V 交流电，再转换为计算机元件和芯片所需的 12 V 或者其他电压——4~5 次转换。此外，散热是一个重要因素。如果 1 瓦的功率使计算机产生一定的热量，至少还要花费 1 瓦的电能来进行散热。你实际上并没有使数据中心“冷却”，而是在排出热能。各大公司都希望将其运营成本降低到 Google、Microsoft 和 Facebook 的水平——它们实现了接近 1.2 PUE 的效率。2011 年，Google 报告一个数据中心在冬季达到了 1.08 的 PUE（Google 2012）。PUE 每降低 1/10，是巨大的竞争优势。

其他一些因素甚至比电源效率更重要。使用更少的电力——例如关闭不用的机器——不管 PUE 多高都是一种改进。聪明的企业致力于实现最佳的单位计算价格。例如，对于互联网搜索，真正的效率来自于从你的金钱中得到最大的 QPS。这可能意味着略过某一代 Intel CPU，因为这些 CPU 尽管更快，但是耗电量的增大却不成比例。

新的经济学还使赠送某种服务更为可行，因为广告收入已经足以支撑业务。许多网站免费提供大部分功能，但是对于“增值”服务额外收费。这种“免费增值”业务模式是吸引了大量客户的有效方式。免费意味着准入门槛很低。一旦用户习惯于使用某种服务。销售增值功能就更容易了。如果广告收入充足，免费用户就不成为负担。

这种新方法显著降低了软件、硬件和电力成本，催生了新的业务模式。这种价格下降可能达到计算成本为零的程度吗？答案可能令你吃惊。

B.5 云计算时代（2010 年至现在）

目前为止的趋势是，每一个时代计算能力都比前一个时代更经济。容量和可靠性的增加造成的成本增加从超线性变为线性，再变为次线性。这方面还有改进的余地吗？计算能力能否变成免费的资源？

许多房东住在“免费出租”的房子里，因为他们有足够的高利润租户，可以支付所居住的房子费用。这种经济思想使云计算成为可能：构建比所需要的更多的计算资源，并将剩余的资源出租。

可用性需求

在这段时间，移动计算变得更容易承受，也更容易获得。智能化手机运行和 PC 一样复杂的操作系统。创建移动应用要求更短的延迟和更高的可靠性。

移动应用创造了更低延迟服务的需求——更快响应的服务。如果移动地图应用需要花费 1 小时才能算出到某个位置的最佳驾车路径，这个应用也就没有什么用处了。应用必须有足够的速度，在 1~2 秒内响应，在实际环境中才能起作用。如果这种计算每秒能够进行数百次，就为新的应用打开了大门：拖动地图上的点，实时计算路径。现在发送的不是一个请求而是数十个请求。因为达到这种效果应用程序的实用性就会大大增加，用户数也会随之增加。所以，容量需求增加了好几个数量级，支持基础设施上也必须有巨大的飞跃。

移动应用的可靠性需求也达到了新的水平。如果所依赖的地图服务在你需要的时候停运，地图应用也就不是很有用了。是的，地图的各个部分可以缓存，但是实时的交通报告没法做到这一点。随着移动应用越来越性命攸关，相关支持服务的可靠性也变得越来越重要。随着可靠性的改进取得飞跃，更多性命攸关的应用成为现实。

成本

当计算在更大的规模上进行时，新经济学应运而生。如果计算规模越大越经济，构建较大基础设施就很有好处。如果你构建一个比需求更大的基础设施，可以开发技术，将闲置的容量“租赁”给其他人。这样你使用的部分比以往更便宜，而不使用的部分转化成利润。如果利润较小，可以抵消基础设施的成本。如果利润足够大，就可以支付整个基础设施的费用。在那个时候，你的计算基础设施就变成“免费”的。如果经营得当，甚至可以拥有负成本的基础设施。想象一下运营像 Amazon 那样的基础设施，却由别人买单，是怎么样的一种感受。再想象一下，如果你试图启动一个新的售书网站，而竞争对手的基础设施是“免费”的，又是一种什么样的感受。这就是推动云计算供应商端的经济愿景。

在云计算时代，规模经济使新的订单成本更加便宜。这产生了足够的余地，使服务的定价低于客户自己所能达到的水平，产生补贴提供商基础设施的附加利润。对运营效率的改进增加了服务提供商的利润，或者使其能以比竞争对手更低的价格提供服务。

要理解客户的云计算需求，必须观察小规模计算系统的相关成本。在小规模计算中，无法利用分布式计算经济的好处。相反，人们必须通过更昂贵的硬件实现可靠性。在小规模计算中，自动化的意义也不大，只会推高运营成本。因为成本得不到分摊，创建自动化会更加昂贵。许多分布式计算需要具备专业知识的人员，而小公司不具备这种条件，因为它们必须雇佣通才，无法承受一位全职人员（或者团队）仅监控系统的一个方面。需要这种专业知识时，使用外部顾问也是很昂贵的。

当小规模计算通过租赁云基础设施上的空间进行时，上述的许多问题会得到缓解。客户能够利用更低的成本和更高的电源效率。云计算能够提供难以维护的服务（如专用存储和网络技术），隐藏了这些系统需要的困难的后台管理。

对于客户来说，还有成本之外的好处。弹性是动态增减资源消费量的能力。对于许多应用程序，可以在短期内征用许多服务器是很有价值的。假定你有一个只持续一周的广告宣传活动。在活动进行中，可能需要几百个 Web 服务器处理流量。在几个小时内获得这么

多的服务器，是一种极其宝贵的能力。大部分公司需要几个月甚至一年才能够安装这么多的服务器，广告机构以前也没有过这种能力。现在，甚至不需要知道是谁建设了数据中心，广告机构就可以得到所有必需的系统。更重要的可能是，在活动结束时，这些服务器可以“归还”给云提供商。用旧的物理硬件方法来实现这一过程是不切实际的。

对于许多公司来说，另一个成本之外的好处是云计算使其他部门可以避开顽固不化或者难以打交道的 IT 部门。单击一次鼠标就可以获得所需的计算资源，而不需要花费几个月和不肯合作或者资金不足的 IT 部门争论，这种能力很有吸引力。我们不得不羞愧地承认，IT 部门确实存在这种情况，这也往往成为人们采用云计算服务的一个理由。

伸缩性和高可用性

满足云计算时代伸缩性和高可用性的新需求，需要新的模式。更低的延迟主要通过更快的存储技术和更快移动信息的方法实现。

在这个时代，SSD 已经取代了磁盘。SSD 更快是因为没有活动部件。不需要等待读磁头移动到磁盘的正确部分，不需要等待磁盘旋转到正确位置。SSD 的每 GB 成本更高，但是总拥有成本较低。假定你需要 10 个数据库服务器副本，提供马力，才足以在要求的延迟下提供服务。使用 SSD 虽然更贵，但是相同的延迟可以用更少的机器提供（总数通常只需要 2~3 台）。毫无疑问，SSD 更贵，但是不会比多出来的 7 台机器更贵。

降低内部通信延迟，也能够降低服务延迟。过去，两台机器之间的信息通过许多技术层次发送。信息被转换为“线路格式”，意味着制作一个用于传输的副本，并将其放在数据包中。然后，数据包通过操作系统的 TCP/IP 层和设备层、网络传输，到达另一台机器进行反向处理。每一步都增加延迟。大部分或所有此类延迟现在都已经通过技术消除，这些技术允许在机器之间进行直接内存访问。有时候，这些技术甚至绕过源或目标机器的 CPU。结果是，在机器之间传递信息的速度几乎和读取本地 RAM 一样。延迟如此之小，以至于底层 RPC 机制不得不重头设计，以全面利用这些新功能。

技术

推动云计算本身的技术是对基础设施进行分段、使一个“租户”不会干扰另一个租户的能力。这就需要从网络技术上实现在非常细的粒度上（如可编程 VLAN 和软件定义网络）对计算机进行分段的能力。这要求将一台大型机器分解为许多较小的机器——虚拟化技术；需要控制面板、API 和各种自助服务式管理功能，使客户能够自行支持，不会给数据中心运营增加过多的劳动成本；需要存储和其他服务以吸引客户。

Amazon 是开发此类系统的第一家公司。Amazon 弹性计算云（Amazon EC2）是第一家租赁其基础设施上的虚拟机以补贴自身成本的公司。“弹性”一词来源于这样一个事实：这种方法可以快速扩展和收缩所使用的资源量。最吸引人的功能是启动新机器的速度很快。更令人痴迷的是在不需要时处置机器的速度。假定你需要 1000 台机器来启动长达一个月的

Web 广告宣传活动。自行安装这么多机器是艰巨的任务，一个月后也难以摆脱这些机器。利用 EC2，只需要单击一下鼠标或者几次 API 调用即可启动所有机器。当活动结束后，释放这些机器也同样简单。

这种“租户”系统刚刚起步。我们现在才刚刚开始理解这种新经济及其衍生品。

B.6 结语

本附录的目标是解释用于提供 Web 服务的各项技术的历史。这里描述的技术组成了第 3 章介绍的平台选择、第 4 章中介绍的 Web 架构和第 5 和第 6 章描述的伸缩性和弹性技术的基础。本书的其他章节反映了使上述基础设施正常工作的运营实践。

计算经济学随着时间的推移而不断变化。更快、更可靠的计算技术在 Web 前时代和第一个 Web 时代中具有超线性的成本。第二个 Web 时代是依靠线性成本曲线实现的。云计算为我们提供了次线性成本曲线。这些变化是利用商品化和标准化、转换到开源软件、通过软件而非硬件提高可靠性以及用更多软件替代劳力密集操作实现的。

计算成本每改善一个数量级，都会开启一个新的应用时代，每个时代的成果都是几年前所无法想象的。1983 年使用 8 位计算机核对支票簿的人，何曾想象到 Facebook 或者 Google 眼镜？

所有人都在猜测云计算之后会出现什么。明天的应用需要更快、延迟更短、更便宜的计算资源。见证这样的发展是激动人心的。

练习

1. 本附录提供了计算技术历史上的 5 个时代：Web 前时代、第一个 Web 时代、.com 炸弹时代、第二个 Web 时代和云计算时代。对于每个时代，描述经济繁荣程度、技术改进和可靠性及计算能力的预期。
2. 拥有整个过程而不为某个步骤或者部分使用外部供应商的实践，称作“垂直集成”。本附录描述了哪些垂直集成的例子？这种集成有何好处？
3. 开源软件对第二个 Web 时代的成熟起到了什么作用？
4. 描述被 ROC 分布计算代替的冗余可靠性级别。
5. 本附录讨论的历史是不可避免的，你同意还是不同意这种观点？提出支持的论据。
6. 云计算之后会迎来什么时代？确定本附录描述的趋势，并根据过去的表现推断未来。

伸缩性术语和概念

生活的本质就是规模巨大的统计非概率性。

——Richard Dawkins

你可能曾经遇到在小数据量下工作得很好，但是随着数据的增加越来越慢的软件。有些系统在数据更多时略慢一些。而其他系统的减速幅度要大得多，往往具有戏剧性。讨论可伸缩性时，有用于描述这种现象的术语。这使我们可以用专门的名词相互沟通。本附录描述了一些基本术语，这些术语以数学的方式描述了相同的概念，最后，本附录还提供关于现代系统是否如预期一样工作的说明。

C.1 恒定、线性和指数伸缩

描述系统性能和伸缩性的术语很多。有 3 个术语常常用到，作为专业系统管理员，你应该善于使用这些术语。它们描述了系统在数据规模增长时的表现：系统不受影响、越来越慢或者大幅度变慢。

- ❑ **恒定伸缩**：不管输入数据的规模如何，性能都没有变化。例如，RAM 中的散列表不管包含 100 个项目还是 1 亿个项目，查找操作总是在固定时间内完成。如果所有系统都有这么快，那就太好了，但是这样的算法很少见。实际上，即使是散列表查找也受限于 RAM 能否容纳这么多数据。
- ❑ **线性伸缩**：随着输入规模的增长，系统速度按比例降低。例如，两倍的数据需要两倍的处理时间。假定某个服务有 10 000 个用户，其身份验证系统对每个请求的身份验证需要花费 10 毫秒。当用户数为 20 000 个时，系统可能花费 60 毫秒。当用户

数为 30 000 个时，该系统可能花费 110 毫秒。每个单位的增长造成相同规模的速度降低（每增加 10 000 个用户，速度降低 50 毫秒）。因此，我们可以将这种系统归类为性能与用户数据库大小呈线性关系的系统。

- ❑ **指数伸缩**：随着输入规模的增长，与系统速度降低不成比例降低。继续身份验证系统的例子，如果增加更多用户造成 10 毫秒、100 毫秒和 10 000 毫秒的响应时间，这可能就是指数伸缩。速度以这种比例降低的系统如果与系统的其他各个部分交互，那就是一场灾难！如果输入规模预计不会变化，在当前规模下这种系统已经足够快了——这种假设有很高的风险。

C.2 大 O 标记法

$O()$ （“大 O 标记法”）用于根据系统对输入规模变化的响应分类。这是更为数学化的系统行为描述方法。 $O()$ 标记法来自于计算机科学。使用字母“O”是因为算法运行时间的增长率被称作“阶”（Order）。下面是系统管理员应该熟悉的常见“大 O”术语。我们已经描述过其中的前三个：

- ❑ $O(1)$ ：**恒定伸缩**。不管输入规模如何，性能都没有变化。
- ❑ $O(n)$ ：**线性伸缩**。运行速度和输入规模的增长等比例降低。
- ❑ $O(n^m)$ ：**指数伸缩**。性能以指数形式劣化。
- ❑ $O(n^2)$ ：**平方伸缩**。性能的劣化与输入规模的平方成正比。平方伸缩是指数伸缩的特例， $m=2$ 。人们说一个系统“指数”伸缩时，实际上往往指的是“平方”伸缩。这种说法十分常见，以至于现在很少听到“平方”这一词，除非某个人明确（或者卖弄学问）地说明性能曲线。
- ❑ $O(\log n)$ ：**对数伸缩**。性能的劣化与输入规模的对数（ \log_2 ）成正比。随着输入规模的增长，系统性能下降趋势趋缓。例如，对分查找算法随着搜索语料库大小的增长而变慢，但是低于线性增长幅度。
- ❑ $O(n \log n)$ ：**对数线性伸缩**。性能劣化的速度快于线性，“加速度”与输入规模的对数（ \log_2 ）成正比。可以将这种情况视为特殊的线性伸缩，随着输入规模增长，性能劣化的速度不断增长但是幅度越来越小。人们往往在引用“对数线性伸缩”时错误地使用“对数伸缩”一词。使用“对数线性”一词能让你听起来像真正的专家。
- ❑ $O(n!)$ ：**阶乘伸缩**。性能的劣化与输入规模的阶乘成正比。换言之，性能劣化很快，规模每增加一个单位，性能的劣化程度比前面每一次增加造成的总和还大！ $O(n!)$ 算法通常是最糟糕的情况。例如，Google PageRank 和 Facebook SocialRank 的优越性能来源于计算机科学研究的一项突破，这项研究就是发明了替代 $O(n!)$ 算法的新算法。

次线性（Sub-Linear）一词指的是比线性更慢的增长，如恒定及对数伸缩。**超线性**

(Super-Linear) 指的是比线性更快的增长, 如指数和阶乘伸缩。

除了描述伸缩性之外, 这些术语还常常用于描述增长率。人们可能将业务所吸引的客户增长描述为线性增长或者指数增长。系统运行时间的增长也可以用类似的术语描述。

和次线性系统相比, 超线性系统听上去很糟糕。为什么不始终使用恒定或者线性的算法呢? 最简单的理由是, 这一级别的算法往往不存在。排序算法需要至少接触每个排序项一次, 这就排除了 $O(1)$ 算法的可能性。有一种 $O(n)$ 排序算法, 但是只对某类数据有效。另一个原因是, 较快的算法往往需要提前进行额外的工作: 例如, 构建索引可以使未来的搜索更快, 但是需要承受索引构建和维护的开销和复杂性。如果系统性能本身已经足够好, 开发、测试和维护这些索引代码的努力可能不值得。

在很小的数值上, 不同阶的系统可能等价。 x 和 x^2 哪一个更大? 直觉的反应是平方数会大于原值。但是, 如果 x 是 0.5, 这一判断就不成立: $0.5^2=0.25$, 小于 0.5。同样, 对于很小的输入, $O(n)$ 算法可能比 $O(n^2)$ 算法慢。而且, $O(n^2)$ 算法尽管对较大规模的输入不是最有效, 但是可能最容易实现。因此, 开发更复杂的算法可能是在浪费时间, 而且风险更大。较复杂的代码更可能出现缺陷。

重要的是, 要记住 $O()$ 的概念是以输入增长趋势为焦点的概括, 而不是针对具体的运行时间的。例如, 两个阶都为 $O(n^2)$ 的系统性能并不完全相同。相同的阶只是表示两个系统在伸缩性上都劣于更低阶的系统 (如 $O(n)$ 系统)。

$O()$ 标记法是最大因素的指标, 而不是所有因素的指标。一个真实系统可能为一个 $O(n)$ 查找所左右, 但是还可能包含许多 $O(1)$ 操作, 甚至一些 $O(n^2)$ 操作, 只是因为其他原因而不将这些操作作为重点考虑。

例如, 一个 API 调用可能在一个授权用户的小型列表上执行 $O(n)$ 查找, 然后将大部分时间花在一个大数据库中的 $O(1)$ 查找上。如果授权用户列表很小 (比如, 校验该用户是 3 个允许使用系统的人之一) 并且在 RAM 中进行, 这和数据库查找相比就无关紧要。因此, 这个 API 调用可以视为 $O(1)$, 而不是 $O(n)$ 。这样的系统可能在很多年中以可接受的速度运行, 但是之后授权用户激增 (可能是因为管理方法改变而为几千名用户授权)。 $O(n)$ 查找突然开始决定 API 调用的运行时间, 在生产系统中, 这种意外总在发生。

虽然大 O 标记法中的 “ O ” 是阶 (Order) 的简写, 在交流中你还可能听到 “阶” 被作为数量级 (order of magnitude) 的简写的情况。数量级和阶相关但是不同。数量级的实际意义是描述规模的数字位数 (数字量级含义是它处于个位、十位、百位还是其他位数)。如果你为一家有 100 名雇员的公司工作, 而你的朋友则在有 1000 名雇员的公司工作, 那么他所在的公司比你所在的公司大一个数量级。你可能听说过某个算法在 “100 万个用户上运行正常, 但是在更大数量级上崩溃。” 的情境, 也可能听说过某个系统处理 “速度阶为 2000 查询 / 秒”, 这是 “大约 2000” 的有趣说法, 暗指该数值是很粗略的估计。

C.3 大 O 标记法的局限性

$O()$ 标记法与算法执行的运算次数有关。但是并不是所有运算花费的时间都一样。如果两个相邻的数据库记录在同一个磁盘数据块中, 读取这两个记录实际上只是读一次磁盘之后, 从缓存整个数据块的 RAM 上读取两次。两个具有相同阶的算法, 一个利用了这一事实, 另一个没有, 它们的性能将差别巨大。

在 Kamp (2010) 对 Varnish HTTP 加速器的描述中, 他解释了这一点: 利用现代 OS 处理磁盘 I/O 的深入理解, 可以得到比简单的 $O()$ 对比好得多的性能。Varnish 可以用 3 台闲置率为 90% 的机器代替 12 台运行竞争对手软件包的超载机器。他写道:

如果操作导致页面错误和缓慢的磁盘操作, $O(\log_2 n)$ 算法有什么好的? 对于大部分相关的数据集, 避免页面错误的 $O(n)$ 甚至 $O(n^2)$ 算法要远远胜出。

改进来自于他构造堆时采用了避免产生虚拟内存页面错误的节点。Kamp 安排数据结构, 使父节点和子节点尽可能处于虚拟内存的同一个页面上。

页面错误发生在程序访问当前不在计算机 RAM 中的虚拟内存时。操作系统暂停进程, 从磁盘读取数据, 在读取完成时恢复进程运行。由于内存以 4KB 的块处理, 第一次访问某个块可能导致页面错误, 但是以后在同一个 4KB 区域内的访问不会导致错误。

页面错误花费的时间可以运行 1000 万条指令。换言之, 花费 10 000 条指令来避免页面处理可以得到 1000 倍的回报。通过小心地组织内存对象的存储位置, Kamp 可以消除许多页面错误。

他以非常技术性的用词详细说明, 大部分计算机科学教育仍然在传授适合于已经消失 30 年的计算机系统的算法设计, 向行业传达这样的信息: “你们错了”。现代 CPU 有许多复杂性, 使性能变得不那么直观。线性访问明显快于随机访问。当指令相互独立时可以并行执行。当多个 CPU 访问内存中的同一部分时, CPU 性能陡降。

例如, 按照行顺序读取数组中的每个元素明显快于列顺序读取。如果行大于 CPU 内存缓存, 后者实际上在每次读取时都会造成缓存“未命中”。前者采用线性的内存读取, CPU 在这种方法下得到优化, 操作更快。即使两种方法都从内存中读取相同的字节数, 前者也更快。

更令人吃惊的是, 读取数组每个元素的循环执行的时间和读取相同数组中每个第 16 元素的循环大致相同。尽管前者进行的操作数量只有 1/16, 但是两个循环造成的 RAM 缓存未命中数量相同。从 RAM 读取内存块到 CPU 的 L1 (1 级) 缓存很慢, 占据两个算法运行时间的绝大部分。前者运行更快是因为有用顺序读取内存的特殊指令。

除了 RAM、虚拟内存和磁盘的问题, 我们还必须考虑线程的效果、多个 CPU 试图访问相同数据、加锁、互斥体等因素。这些因素使性能的问题更加模糊。没有基准测试, 你就无法真正说出算法的速度。 $O()$ 标记法变成一般方针而非绝对标准。虽然如此, $O()$ 标记法在描述系统以及在与其它系统管理员的沟通中仍然很有用。因此, 对这些术语的理解是现代系统管理员必不可少的。

模板和示例

D.1 设计文档模板

下面是 13.2 节描述的简单设计文档模板。

标题：

日期：

作者：（添加作者、请链接到他们的电子邮件地址）

审核人：（添加审核人，请链接到他们的电子邮件地址）

批准人：（添加批准人，请链接到他们的电子邮件地址）

修订号：

状态：（草稿、审核中、已批准、进行中）

执行摘要：

（2~4 句，解释项目）

目标：

（描述所解决问题的项目列表）

非目标：

（描述项目局限性的项目列表）

背景：

（理解设计所需的术语和历史）

概要设计：

（设计的简短概述，框图很有帮助）

详细设计：

(详细介绍设计，通常是自顶向下的描述)

考虑的替代方案：

(考虑和拒绝的替代方案，包括各个方案的优劣)

安全考虑因素：

(安全、隐私分支以及缓解这些忧虑的方法)

D.2 设计文档示例

下面是 13.2 节中描述的设计文档的一个例子。

为了说明设计文档的原则，我们已经创建了一个设计文档样板，包含大部分内容：

标题：新监控系统

日期：2014-07-19

作者：Strata Chalup src@example.com 和 Tom Limoncelli tal@example.com

审核人：Joe, Mary, Jane

批准人：安全运营部门 2014-08-03, Chris (构建系统技术负责人) 2014-08-04, Sara (运营主管) 2014-08-10

修订号：1.0

状态：已批准

执行摘要：

为设备创建监控系统，支持通过传呼机的实时警报，部署到生产群集。

目标：

用于我们的网络的监控系统：

- ☐ 可以监控至少 10000 台设备，每台设备 500 个属性，每分钟收集一次。
- ☐ 必须支持通过 SMS 的实时警报。
- ☐ 必须保留 30 年的数据。

非目标：

- ☐ 外部系统的监控。
- ☐ 在系统中加入聊天功能。

背景：

历史上，我们已经使用了内部制作的监控脚本，但是必须根据平台的变化不断调整。我们希望有标准化的监控方法，不需要在增加平台时花费时间开发。

概要设计：

我们的计划是使用具有远程收集器的 Bosun。该服务器将成为服务器硬件和 Linux 的当前企业标准配置。该服务器命名为 bosun01.example.com，将放置在菲尼克斯和多伦多的数据中心。监控配置将保存在 Git 存储库“sysadmin configs”的最高级目录/monitor 下。

详细设计：**服务器配置：**

Dell 720XD，64G RAM，有 8 个硬盘组成 RAID 6 配置

Debian Linux（标准企业“服务器配置”）

Bosun 软件包名称为“bosun-server”

- ☐ 任何被监控系统将安装“bosun-client”软件包。
- ☐ 备份将使用标准企业备份机制，在夜间进行。
- ☐ 完整文档将包含运营任务的描述，如添加新监控收集器和警报规则。

成本规划：

- ☐ 初始成本【服务器成本】。
- ☐ 软件是开源的。
- ☐ 一位全职工程师在 3 周内花费一半的时间设置监控并启动。

批准后的时间安排：

1. 1 周采购硬件。
2. 2 天安装和测试。
3. 在第 3 周上线。

考虑的替代方案：

考虑了 Zabbix 和 SkunkStorm，但是 Bosun 的功能集更符合项目目标。【比较图表的链接】

特殊约束：

Bosun 在一个密码保险箱中保存 SNMP“社群字符串”，这实际上是密码。存储方法已经过安全运营部门的审核和批准。

D.3 事后剖析模板样板

下面是 14.3.2 节中描述的一个简单事后剖析模板。

标题：

报告状态：

执行摘要：

列出发生的情况，受到影响的人以及避免未来发生的关键建议（特别是需要预算或者高管批准的部分）

运行中断描述：

运行中断的总体描述，从技术方面入手。

受影响的用户：

受影响的人。

开始日期 / 时间：

结束日期 / 时间：

持续时间：

时间轴：

每分钟的时间轴，由系统日志、聊天日志、电子邮件和其他可用资源组成。

归因分析：

导致运行中断的根源是什么？

哪些方面做得好？

处置得当方面的项目列表。这是感谢提供超出预期帮助的所有人的好机会。

哪些方面可以做得更好？

一个项目列表，列出可以采取哪些行动改进服务恢复速度、所使用的技术等。

建议：

在未来避免运行中断的建议列表。每个建议都应该是可以付诸行动、可计量的。好的例子：“监控数据库服务器的磁盘空间，在可用空间小于 20% 时警报”。不好的例子：“改进监控”。为每个建议提交一个缺陷 / 功能请求，在此列出缺陷 ID。

所涉人员姓名：

涉及运行中断解决的人员列表。

推荐读物

要实现伟大的目标需要两个条件：一个计划，以及不那么足够的时间。

—Leonard Bernstein

DevOps:

- ❑ 《The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win》(Kim et al. 2013)
传授 DevOps 三条道路的虚构故事。
- ❑ 《Building a DevOps Culture》(Walls, 2013)
建设 DevOps 文化的策略，包括调整激励措施，定义有意义和可实现的目标，创建支持性团队环境。
- ❑ 《Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation》(Humble & Farley, 2010)
关于服务交付平台的权威书籍。
- ❑ 《Release It!: Design and Deploy Production-Ready Software》(Nygard, 2007)
实现第 11 章中许多想法的详细介绍和示例。
- ❑ 《Blameless PostMortems and a Just Culture》(Allspaw, 2009)
事后剖析的理论与实践。
- ❑ 《A Mature Role for Automation》(Allspaw, 2012c)
解释“自动化任何操作!”不是好主意的原因，以及替代方案。
- ❑ 《Each Necessary, But Only Jointly Sufficient》(Allspaw, 2012a)
“根源分析”和“5 个为什么”的神话和局限。

ITIL: 《Scalability Rules: 50 Principles for Scaling Web Sites》(Abbott & Fisher, 2009)

- 《Owning ITIL: A Skeptical Guide for Decision-Makers》(England, 2009)

ITIL 的简介, 包含了如何实现 ITIL 的直率建议。

理论: 《The Design and Implementation of the FreeBSD Operating System》(Kernighan & Pike, 1999)

- 《In Search of Certainty: The Science of Our Information Infrastructure》(Burgess, 2013)

分布式系统的物理学。

- 《Promise Theory: Principles and Applications》(Burgess & Bergstra, 2014)

配置管理和更改管理的理论。

经典 Google 文章: 《The UNIX Programming Environment》(Kernighan & Pike, 1978)

- 《The Anatomy of a Large-Scale Hypertextual Web Search Engine》(Brin & Page, 1998)

第一篇描述 Google 搜索引擎的文章。

- 《Web Search for a Planet: The Google Cluster Architecture》(Barroso, Dean & Hölzle, 2003)

Google 第一篇揭示如何使用具备容错软件的商品化级 PC 设计群集的文章。

- 《The Google File System》(Ghemawat, Gobioff & Leung 2003)

- 《MapReduce: Simplified Data Processing on Large Clusters》(Dean & Ghemawat, 2004)

- Bigtable: A Distributed Storage System for Structured Data》(Chang 等, 2006)

- 《The Chubby Lock Service for Loosely-Coupled Distributed Systems》(Burrows, 2006)

- 《Spanner: Google's Globally-Distributed Database》(Corbett 等, 2012)

- 《The Tail at Scale》(Dean & Barroso, 2013)

缩短延迟意味着调整最后一个百分位数。

- 《Failure Trends in a Large Disk Drive Population》(Pinheiro, Weber & Barroso, 2007)

硬盘故障的纵向研究。

- 《DRAM Errors in the Wild: A Large-Scale Field Study》(Schroeder, Pinheiro & Weber, 2009)

DRAM 故障的纵向研究。

经典 Facebook 文章: 《The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise》(Abbott & Fisher, 2009)

- 《Cassandra: A Decentralized Structured Storage System》(Lakshman & Malik, 2010)

可伸缩性:

- 《The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise》(Abbott & Fisher, 2009)

人、过程、技术可伸缩性的详细技术分类及讨论。

- 《Scalability Rules: 50 Principles for Scaling Web Sites》(Abbott & Fisher, 2011)
精简版本, 专注于技术策略和技术。

UNIX 内部结构:

- 《The Design and Implementation of the FreeBSD Operating System》(McKusick, Neville-Neil & Watson, 2014)
这是深入 UNIX 类操作系统工作原理的最佳书籍。虽然例子都采用了 FreeBSD, 但是 Linux 用户可以从该书提供的理论和技术细节中得益。

UNIX 系统编程:

- 《The UNIX Programming Environment》(Kernighan & Pike, 1984)
- 《Advanced Programming in the UNIX Environment》(Stevens & Rago, 2013)
- 《UNIX Network Programming》(Stevens, 1998)

网络协议:

- 《TCP/IP Illustrated, Volume 1: The Protocols》(Stevens & Fall, 2011)
- 《TCP/IP Illustrated, Volume 2: The Implementation》(Wright & Stevens, 1995)
- 《TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols》(Stevens & Wright, 1996)

后记 *Postscript*

我们寻找让自己奔走的事物。

——Pakled Captain Grebnedlog

我们想要更好的世界。

我们想要一个食物更安全、味道更好的世界。

我们想要一个按时交付的世界。

我们想要一个能够更快、更高效地访问机动车辆管理局的世界。

我们想要一个人际关系更密切、更有意义、更有爱的世界。

我们想要一个工作更快乐的世界。

我们想要一个科幻小说中辉煌的幻想变成每个人都能拥有的产品及体验的世界。

我们想要一个没有战争、饥饿、贫穷和仇恨的世界。

我们想要一个乐观、科学和真理战胜悲观、无知和谎言的世界。

我们想要一个每个人都为了全世界的进步而一同工作的世界。

作为本书的作者，我们出生于计算机可以放在桌面、甚至口袋的年代。青少年时代，我们看到了计算机的兴起，深深感觉到科技将会拯救世界。长大成人之后，我们见证了科技改变世界，使世界变得更小，使我们能做到父母从未梦想过的事情。

现在，我们认识到计算机和软件只是时代的一个缩影。想让世界变得更好，需要联系所有部件、联系所有人的运营实践，运行、维护系统并保持正常状态。

人们有时说软件“吞噬”某个行业，打破原有格局并进行彻底变革。发生这种情况时，决定成败的是运营实践。

制造和分发食品的后勤工作现在是一个软件职能。从农场到餐桌，更好的软件可以利用较少的资源实现更多的产出，在合适的时候收割和运输。因为运营实践的成功，我们吃到了比以往更新鲜、更便宜、种类更多样的食品。

在线购物这样简单的行为也需要协同工作的组织链条：原材料供应商、工厂、配送、销售、市场、购买、物流和交付。每一项工作都在软件中体现，运行是否有效也取决于所涉及的运营实践。

改进的周期更快地将新思路 and 科技带入市场，加速了新产品的创造。新思路又引出更新的思路。谁曾经想象到，有一天我们能够使用手机运行应用，拍张照片就能将钱存入支票，按下一个按钮就能倒车，向不稳定结构上投射愤怒的小鸟？

当运营处理得当时，我们的工作更加快乐。我们消除了重大升级的恐惧和不确定性，不再需要在奇怪的时间里完成某些任务，偷走我们的睡眠时间、打搅我们的雅兴。伤害业余时间人际关系的工作压力一去不复返，生活变得更加幸福。我们有更多的时间去生活、去爱，有更多的自由时间去帮助其他人。我们的中心充满幸福和爱，促使我们与他人分享。

下面这些概念还处于早期，其名称和定义仍然不断发展：云计算、分布式计算、DevOps、SRE、Web 和物联网。我们站在山脚下仰望，憧憬着未来。

即使遵循本书中的所有建议，也不能解决全世界的问题，不能消除贫穷、使食物的味道更好。在我们编写本书时，这些建议正在变得过时。

但是这是一个起点。

我们希望本书带给你好的思路、值得尝试的有趣想法、一个出发点。我们仅仅收集了从其他人那里学到的、读到和听说的，以及有幸体验过的好想法。我们是书记员，希望将这些想法和经验转化成表达其精华的语言，而不会有错误的表达或者遗漏重要的内容。如果我们没有做到，在此先向读者道歉。

我们刚刚起步，本书只是一种声音，其余的一切都取决于你。利用在这里学到的，建设一个更好的世界。

参考文献 Bibliography

- Abbott, M., & Fisher, M. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*, Pearson Education.
- Abbott, M., & Fisher, M. (2011). *Scalability Rules: 50 Principles for Scaling Web Sites*, Pearson Education.
- Abts, D., & Felderman, B. (2012). A guided tour through data-center networking, *Queue* 10(5): 10:10–10:23.
<http://queue.acm.org/detail.cfm?id=2208919>
- Adya, A., Cooper, G., Myers, D., & Piatek, M. (2011). Thialfi: A client notification service for internet-scale applications, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pp. 129–142.
<http://research.google.com/pubs/pub37474.html>
- Allspaw, J. (2009). Blameless postmortems and a just culture.
<http://codeascraft.com/2012/05/22/blameless-postmortems>.
- Allspaw, J. (2012a). Each necessary, but only jointly sufficient.
<http://www.kitchensoap.com/2012/02/10/each-necessary-but-only-jointly-sufficient>.
- Allspaw, J. (2012b). Fault injection in production, *Queue* 10(8): 30:30–30:35.
<http://queue.acm.org/detail.cfm?id=2353017>
- Allspaw, J. (2012c). A mature role for automation.
<http://www.kitchensoap.com/2012/09/21/a-mature-role-for-automation-part-i>
- Anderson, C. (2012). Idea five: Software will eat the world, *Wired*.
http://www.wired.com/business/2012/04/ff_andreessen/5
- Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing, *Communications of the ACM* 53(4): 50–58.
<http://cacm.acm.org/magazines/2010/4/81493-a-view-of-cloud-computing>
- Barroso, L. A., Dean, J., & Hölzle, U. (2003). Web search for a planet: The Google cluster

- architecture, *IEEE Micro* 23(2): 22–28.
<http://research.google.com/archive/googlecluster.html>
- Barroso, L. A., Clidaras, J., & Hölzle, U. (2013). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed., Morgan and Claypool Publishers.
<http://research.google.com/pubs/pub41606.html>
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for agile software development.
<http://agilemanifesto.org>
- Black, B. (2009). EC2 origins.
<http://blog.b3k.us/2009/01/25/ec2-origins.html>
- Brandt, K. (2014). OODA for sysadmins.
<http://blog.serverfault.com/2012/07/18/ooda-for-sysadmins>
- Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine, *Proceedings of the Seventh International Conference on World Wide Web 7, WWW7*, Elsevier Science Publishers, Amsterdam, Netherlands, pp. 107–117.
<http://dl.acm.org/citation.cfm?id=297805.297827>
- Burgess, M. (2013). *In Search of Certainty: The Science of Our Information Infrastructure*, Createspace Independent Publishing.
- Burgess, M., & Bergstra, J. (2014). *Promise Theory: Principles and Applications*, On Demand Publishing, Create Space.
- Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, USENIX Association, Berkeley, CA, pp. 335–350.
<http://research.google.com/archive/chubby.html>
- Candea, G., & Fox, A. (2003). Crash-only software, *HotOS*, pp. 67–72.
<https://www.usenix.org/conference/hotos-ix/crash-only-software>
- Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., & Gruber, R. E. (2006). Bigtable: A distributed storage system for structured data, *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 205–218.
- Chapman, B. (2005). Incident command for IT: What we can learn from the fire department., *LISA*, USENIX.
<http://www.greatcircle.com/presentations>
- Cheswick, W. R., Bellovin, S. M., & Rubin, A. D. (2003). *Firewalls and Internet Security: Repelling the Wiley Hacker*, Addison-Wesley.
http://books.google.com/books?id=_ZqIh0IbcrgC
- Clos, C. (1953). A study of non-blocking switching networks, *The Bell System Technical Journal* 32(2): 406–424.
<http://www.alcatel-lucent.com/bstj/vol32-1953/articles/bstj32-2-406.pdf>
- Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y.,

- Szymaniak, M., Taylor, C., Wang, R., & Woodford, D. (2012). Spanner: Google's globally-distributed database, *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, USENIX Association, Berkeley, CA, pp. 251-264.
<http://research.google.com/archive/spanner.html>
- Dean, J. (2009). Designs, lessons and advice from building large distributed systems.
<http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>
- Dean, J., & Barroso, L. A. (2013). The tail at scale, *Communications of the ACM* 56(2): 74-80.
<http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale>
- Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters, *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association.
- Debois, P. (2010a). Jedi4Ever blog.
<http://www.jedi.be/blog>
- Debois, P. (2010b). DevOps is a verb. Slideshare of talk.
<http://www.slideshare.net/jedi4ever/devops-is-a-verb-its-all-about-feedback-13174519>
- Deming, W. (2000). *Out of the Crisis*, Massachusetts Institute of Technology, Center for Advanced Engineering Study.
- DevOps-Toolchain. (2010). A set of best practices useful to those practicing DevOps methodology.
<http://code.google.com/p/devops-toolchain/wiki/BestPractices>
- Dickson, C. (2013). A working theory of monitoring, presented as part of the 27th Large Installation System Administration Conference, USENIX, Berkeley, CA.
<https://www.usenix.org/conference/lisa13/working-theory-monitoring>
- Edwards, D. (2010). DevOps is not a technology problem.
<http://dev2ops.org/2010/11/devops-is-not-a-technology-problem-devops-is-a-business-problem>
- Edwards, D. (2012). Use DevOps to turn IT into a strategic weapon.
<http://dev2ops.org/2012/09/use-devops-to-turn-it-into-a-strategic-weapon>
- Edwards, D., & Shortland, A. (2012). Integrating DevOps tools into a service delivery platform.
<http://dev2ops.org/2012/07/integrating-devops-tools-into-a-service-delivery-platform-video>
- England, R. (2009). *Owning ITIL: A Skeptical Guide for Decision-Makers*, Two Hills.
- Fitts, P. (1951). Human engineering for an effective air navigation and traffic-control system, *Technical Report ATI-133954*, Ohio State Research Foundation.
<http://www.skybrary.aero/bookshelf/books/355.pdf>
- Flack, M., & Wiese, K. (1977). *The Story about Ping*, Picture Puffin Books, Puffin Books.
- Gallagher, S. (2012). Built to win: Deep inside Obama's campaign tech.
<http://arstechnica.com/information-technology/2012/11/built-to-win-deep-inside-obamas-campaign-tech>

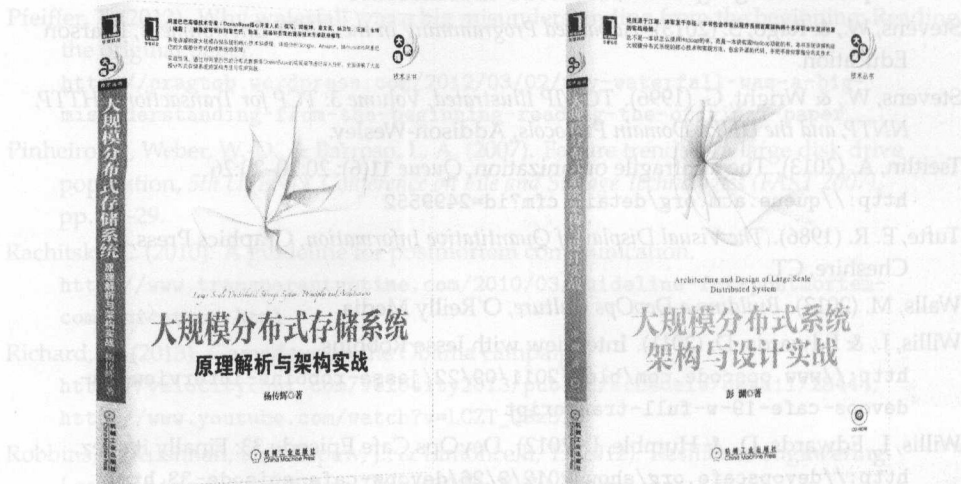
- Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03*, ACM, New York, NY, pp. 29–43.
<http://doi.acm.org/10.1145/945445.945450>
- Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, *SIGACT News* 33(2): 51–59.
<http://doi.acm.org/10.1145/564585.564601>
- Google. (2012). Efficiency: How we do it.
<http://www.google.com/about/datacenters/efficiency/internal>
- Gruver, G., Young, M., & Fulghum, P. (2012). *A Practical Approach to Large-Scale Agile Development: How HP Transformed HP LaserJet FutureSmart Firmware*, Addison-Wesley.
- Haynes, D. (2013). Understanding CPU steal time: When should you be worried.
<http://blog.scoutapp.com/articles/2013/07/25/understanding-cpu-steal-time-when-should-you-be-worried>.
- Hickstein, J. (2007). Sysadmin slogans.
<http://www.jxh.com/slogans.html>
- Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Longman, Boston, MA.
- Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional.
- Jacob, A. (2010). Choose your own adventure: Adam Jacob on DevOps.
<http://www.youtube.com/watch?v=Fx80BeNmaww>
- Kamp, P.-H. (2010). You're doing it wrong, *Communications of the ACM* 53(7): 55–59.
<http://queue.acm.org/detail.cfm?id=1814327>
- Kartar, J. (2010). What DevOps means to me.
<http://www.kartar.net/2010/02/what-devops-means-to-me>
- Kernighan, B., & Pike, R. (1984). *The UNIX Programming Environment*, Prentice Hall.
- Kernighan, B., & Plauger, P. (1978). *The Elements of Programming Style*, McGraw-Hill.
- Kim, G., Behr, K., & Spafford, G. (2013). *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, IT Revolution Press.
<http://books.google.com/books?id=mqXomAEACAAJ>
- Klau, R. (2012). How Google sets goals: OKRs.
<http://gv.com/1322>
- Krishnan, K. (2012). Weathering the unexpected, *Queue* 10(9): 30:30–30:37.
<http://queue.acm.org/detail.cfm?id=2371516>
- Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system, *SIGOPS Operating Systems Review* 44(2): 35–40.
<http://doi.acm.org/10.1145/1773912.1773922>
- Lamport, L. (2001). Paxos made simple, *SIGACT News* 32(4): 51–58.
<http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf>
- Lamport, L., & Marzullo, K. (1998). The part-time parliament, *ACM Transactions on Computer Systems* 16: 133–169.
- Lee, J. D., & See, K. A. (2004). Trust in automation: Designing for appropriate reliance, *Human Factors* 46(1): 50–80.

- Letuchy, E. (2008). Facebook Chat.
https://www.facebook.com/note.php?note_id=14218138919
- Levinson, M. (2008). *The Box: How the Shipping Container Made the World Smaller and the World Economy Bigger*, Princeton University Press.
- Levy, S. (2012). Google throws open doors to its top-secret data center, *Wired* 20(11).
<http://www.wired.com/wiredenterprise/2012/10/ff-inside-google-data-center/all>
- Limoncelli, T. A. (2005). *Time Management for System Administrators*, O'Reilly and Associates.
- Limoncelli, T. (2012). Google DiRT: The view from someone being tested, *Queue* 10(9): 35:35–35:37.
<http://queue.acm.org/detail.cfm?id=2371516#sidebar>
- Limoncelli, T. A., Hogan, C., & Chalup, S. R. (2015). *The Practice of System and Network Administration*, 3rd ed., Pearson Education.
- Link, D. (2013). Netflix and stolen time.
<http://blog.sciencelogic.com/netflix-steals-time-in-the-cloud-and-from-users/03/2011>
- Madrigal, A. C. (2012). When the nerds go marching in: How a dream team of engineers from Facebook, Twitter, and Google built the software that drove Barack Obama's reelection.
<http://www.theatlantic.com/technology/archive/2012/11/when-the-nerds-go-marching-in/265325>
- McKinley, D. (2012). Why MongoDB never worked out at Etsy.
<http://mcfunley.com/why-mongodb-never-worked-out-at-etsy>
- McKusick, M. K., Neville-Neil, G., & Watson, R. N. (2014). *The Design and Implementation of the FreeBSD Operating System*, Prentice Hall.
- Megiddo, N., & Modha, D. S. (2003). ARC: A self-tuning, low overhead replacement cache, *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST '03*, USENIX Association, Berkeley, CA, pp. 115–130.
<https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>
- Metz, C. (2013). Return of the Borg: How Twitter rebuilt Google's secret weapon, *Wired*.
<http://www.wired.com/wiredenterprise/2013/03/google-borg-twitter-mesos/all>
- Nygard, M. T. (2007). *Release It!: Design and Deploy Production-Ready Software*, Pragmatic Bookshelf.
- Oppenheimer, D. L., Ganapathi, A., & Patterson, D. A. (2003). Why do Internet services fail, and what can be done about it?, *USENIX Symposium on Internet Technologies and Systems*.
- Parasuraman, R., Sheridan, T. B., & Wickens, C. D. (2000). A model for types and levels of human interaction with automation, *Transactions on Systems, Man, and Cybernetics Part A* 30(3): 286–297.
<http://dx.doi.org/10.1109/3468.844354>
- Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox,

- A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., & Treuhaft, N. (2002). Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies, *Technical Report UCB/CSD-02-1175*, EECS Department, University of California, Berkeley, CA.
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2002/5574.html>
- Pfeiffer, T. (2012). Why waterfall was a big misunderstanding from the beginning: Reading the original paper.
<http://pragtoob.wordpress.com/2012/03/02/why-waterfall-was-a-big-misunderstanding-from-the-beginning-reading-the-original-paper>
- Pinheiro, E., Weber, W.-D., & Barroso, L. A. (2007). Failure trends in a large disk drive population, *5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pp. 17–29.
- Rachitsky, L. (2010). A guideline for postmortem communication.
<http://www.transparentuptime.com/2010/03/guideline-for-postmortem-communication.html>
- Richard, D. (2013). Gamedays on the Obama campaign.
<http://velocityconf.com/velocity2013/public/schedule/detail/28444>;
http://www.youtube.com/watch?v=LCZT_Q3z520
- Robbins, J., Krishnan, K., Allspaw, J., & Limoncelli, T. (2012). Resilience engineering: Learning to embrace failure, *Queue* 10(9): 20:20–20:28.
<http://queue.acm.org/detail.cfm?id=2371297>
- Rockwood, B. (2013). Why SysAdmin's can't code.
<http://cuddletech.com/?p=817>
- Rossi, C. (2011). Facebook: Pushing millions of lines of code five days a week.
<https://www.facebook.com/video/video.php?v=10100259101684977>
- Royce, D. W. W. (1970). Managing the development of large software systems: Concepts and techniques.
- Schlossnagle, T. (2011). Career development.
<http://www.youtube.com/watch?v=y0mHo7SMCQk>
- Schroeder, B., Pinheiro, E., & Weber, W.-D. (2009). DRAM errors in the wild: A large-scale field study, *SIGMETRICS*.
- Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013). Omega: Flexible, scalable schedulers for large compute clusters, *SIGOPS European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, pp. 351–364.
<http://research.google.com/pubs/pub41684.html>
- Seven, D. (2014). Knightmare: A DevOps cautionary tale.
<http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale>
- Siegler, M. (2011). The next 6 months worth of features are in Facebook's code right now (but we can't see).
<http://techcrunch.com/2011/05/30/facebook-source-code>
- Spear, S., & Bowen, H. K. (1999). Decoding the DNA of the Toyota production system, *Harvard Business Review*.
- Spolsky, J. (2004). Things you should never do, Part I, *Joel on Software*, Apress.
<http://www.joelonsoftware.com/articles/fog0000000069.html>

- Stevens, W. (1998). *UNIX Network Programming: Interprocess Communications, UNIX Networking Reference Series, Vol. 2*, Prentice Hall.
- Stevens, W. R., & Fall, K. (2011). *TCP/IP Illustrated, Volume 1: The Protocols*, Pearson Education.
<http://books.google.com/books?id=a230An5i8R0C>
- Stevens, W., & Rago, S. (2013). *Advanced Programming in the UNIX Environment*, Pearson Education.
- Stevens, W., & Wright, G. (1996). *TCP/IP Illustrated, Volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison-Wesley.
- Tseitlin, A. (2013). The antifragile organization, *Queue* 11(6): 20:20–20:26.
<http://queue.acm.org/detail.cfm?id=2499552>
- Tufte, E. R. (1986). *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT.
- Walls, M. (2013). *Building a DevOps Culture*, O'Reilly Media.
- Willis, J., & Edwards, D. (2011). Interview with Jesse Robbins.
<http://www.opscode.com/blog/2011/09/22/jesse-robbins-interview-on-devops-cafe-19-w-full-transcript>
- Willis, J., Edwards, D., & Humble, J. (2012). DevOps Cafe Episode 33: Finally it's Jez.
<http://devopscafe.org/show/2012/9/26/devops-cafe-episode-33.html>
- Wright, G., & Stevens, W. (1995). *TCP/IP Illustrated, Volume 2: The Implementations*, Pearson Education.
- Yan, B., & Kejariwal, A. (2013). A systematic approach to capacity planning in the real world.
<http://www.slideshare.net/arankejariwal/a-systematic-approach-to-capacity-planning-in-the-real-world>

推荐阅读



大规模分布式存储系统：原理解析与架构实战

作者：杨传辉 ISBN: 978-7-111-43052-0 定价：59.00元

阿里巴巴高级技术专家（OceanBase核心开发人员）撰写，阳振坤、章文嵩、杨卫华、汪源、余锋（褚霸）、赖春波等来自阿里、新浪、网易和百度的资深技术专家联袂推荐

系统讲解构建大规模存储系统的核心技术和原理，详细分析Google、Amazon、Microsoft和阿里巴巴的大规模分布式存储系统的原理。

实战性强，通过对阿里巴巴的分布式数据库OceanBase的实现细节进行深入分析，完整讲解了大规模分布式存储系统的架构方法与应用实践。

大规模分布式系统架构与设计实战

作者：彭渊 ISBN: 978-7-111-45503-5 定价：59.00元

绝技源于江湖、将军发于卒伍，

本书包含作者从程序员到首席架构师十多年职业生涯所经历的实战经验

这不是一本讲怎么使用Hadoop的书，而是一本讲实现Hadoop功能的书，本书系统讲解构建大规模分布式系统的核心技术和实现方法，包含开源的代码，手把手教你掌握分布式技术。

推荐阅读



云计算：概念、技术与架构

作者：Thomas Erl 等 ISBN: 978-7-111-46134-0 定价：69.00元



企业应用架构模式

作者：Martin Fowler ISBN: 978-7-111-30393-0 定价：59.00元



设计模式：可复用面向对象软件的基础

作者：Erich Gamma 等 ISBN: 7-111-07575-2 定价：35.00元



深入理解云计算：基本原理和应用程序编程技术

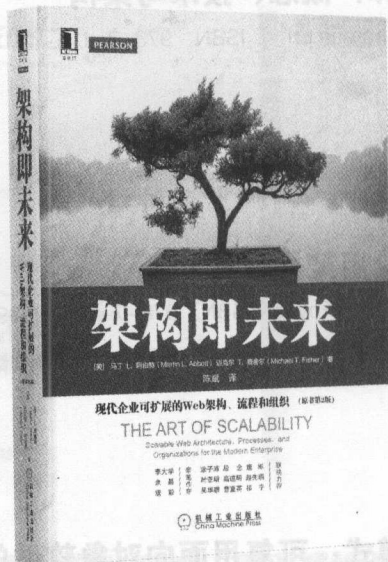
作者：拉库马·布亚 等 ISBN: 978-7-111-49658-8 定价：69.00元



云计算与分布式系统：从并行处理到物联网

作者：Kai Hwang 等 ISBN: 978-7-111-41065-2 定价：85.00元

推荐阅读



架构即未来：现代企业可扩展的Web架构、流程和组织（原书第2版）

作者：马丁 L. 阿伯特 等 ISBN：978-7-111-53264-4 定价：99.00元

本书深入浅出地介绍了大型互联网平台的技术架构，并从多个角度详尽地分析了互联网企业的架构理论和实践，是架构师和CTO不可多得的实战手册。

——唐彬 易宝支付CEO及联合创始人

任何一个持续成长的公司最终都需要解决系统、组织和流程的扩展性问题。本书汇聚了作者从eBay、VISA、Salesforce.com到Apple超过30年的丰富经验，全面阐释了经过验证的信息技术扩展方法，对所需要掌握的产品和服务的平滑扩展做了详尽的论述，并在第1版的基础上更新了扩展的策略、技术和案例。

针对技术和非技术的决策者，马丁·阿伯特和迈克尔·费舍尔详尽地介绍了影响扩展性的各个方面，包括架构、过程、组织和技术。通过阅读本书，你可以学习到以最大化敏捷性和扩展性来优化组织机构的新策略，以及对云计算（IaaS/PaaS）、NoSQL、DevOps和业务指标等的见解。而且利用其中的工具和建议，你可以系统地清除扩展性道路上的障碍，在技术和业务上取得前所未有的成功。



托马斯 A. 利蒙切利

(Thomas A. Limoncelli)

国际知名作家、演说家和系统管理专家。现任 Stack Exchange 公司 SRE，之前曾在 Google NYC 担任 Blog Search（博客搜索）、Ganeti 和各种内部企业 IT 服务项目的 SRE。著有《Time Management for System Administrators》和《The Practice of System and Network Administration》。



斯特拉塔 R. 查卢普

(Strata R. Chalup)

Virtual.Net 公司的所有者和高级咨询师，有超过 25 年的硅谷从业经验，专注于企业的 IT 战略、最佳实践和可扩展基础设施，客户包括 Apple、Sun、Cisco、McAfee 和 Palm 等。



克里斯蒂娜 J. 霍根

(Christina J. Hogan)

资深系统管理专家、系统架构师、安全顾问，在系统管理和网络工程方面有 20 余年从业经验。她也参与项目管理、客户管理和人员管理。

The Practice of Cloud System Administration

Designing and Operating Large Distributed Systems, Volume 2

“这本书中囊括了大量极具深度和思想的实践，看后令人印象深刻。”

—— Win Treese, 《Designing Systems for Internet Commerce》合著者

本书重点关注分布式与云计算，并为系统管理员带来第一手的DevOps/SRE资料。本书全方位地介绍大规模分布式系统的设计和运营，不仅涵盖分布式系统架构、应用和设计原则的理论知识，而且包含Google、Etsy、Twitter、Facebook、Netflix、Amazon等大型技术公司的成功案例分析，能为系统管理员提供有益指导。

设计和构建现代网络与分布式系统

- 介绍大型系统设计的基础知识
- 了解云管理的新软件工程影响
- 使系统能适应失效、增长和动态伸缩
- 实现DevOps原则和文化变革
- 选择IaaS/PaaS/SaaS和虚拟平台

通过最新的DevOps/SRE策略运营和运行系统

- 零停机升级生产系统
- 自动化的概念与应用
- 提高正常运行时间的最佳实践
- 分布式系统应用不同系统管理技术的原因
- 识别和解决弹性问题

评估和评价团队的运营效能

- 管理科学的持续改进过程
- 一个可以立即使用的评估系统



 Pearson
www.pearson.com

vmware PRESS



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/云计算

ISBN 978-7-111-54160-8



9 787111 541608 >

定价: 99.00元